TYPESCRIPT

Подробное Руководство 4.4

TypeScript

Подробное Руководство

Книга и документация в одном

Дата последнего обновления: 10/11/2021

Содержание

Глава 00

Что такое TypeScript и для чего он нужен

Поскольку постижение всего нового занимает много драгоценного времени и сил, то человек, прежде чем сделать выбор, должен взвесить все "за" и "против". Чем больше у него сопряженного опыта, тем легче будет принять решение. Это означает, что недавно присоединившимся к *JavaScript* сообществу людям будет полезнее начать свое погружение в *TypeScript* с подтверждения правильности своего выбора. Кроме того, данная глава сможет ответить на многие вопросы опытных разработчиков, которые обдумывают использование данного языка при создании очередного проекта. Поэтому обойдемся без затягивания и начнем по порядку разбирать самые часто возникающие вопросы связанные с *TypeScript*.

[00.0] Определение

TypeScript — это язык программирования со статической типизацией, позиционирующий себя как язык, расширяющий возможности *JavaScript*.

Typescript код компилируется в *JavaScript* код, который можно запускать как на клиентской стороне (браузер), так и на стороне сервера (*nodejs*). Качество сгенерированного кода сопоставимо с кодом, написанным профессиональным разработчиком с большим стажем. Мультиплатформенный компилятор *TypeScript* отличается высокой скоростью компиляции и публикуется под лицензией *Apache*, а в его создании принимают участие разработчики со всего мира, что привело к традиции выпускать новую версию каждые два месяца. Несмотря на такую периодичность, версии

долгое время сохраняют совместимость, а по истечению долгого времени устаревшее поведение остается доступным при активизации специальных флагов компилятора. Поэтому не стоит опасаться, что проект будет лишен новых возможностей языка из-за версионных различий *TypeScript*. Ну а большое количество разработчиков компилятора означает, что с каждой новой версией компилятор получает обновления, касающиеся всех направлений, будь то синтаксические конструкции или оптимизации скорости компиляции и сборки проекта.

[00.1] История TypeScript

Разработчиком языка *TypeScript* является *Андерс Хейлсберг*, также известный как создатель языков *Turbo Pascal*, *Delphi* и *C#*. С того момента, когда компания *MicroSoft** анонсировал данный язык в 2012 году, *TypeScript* не перестает развиваться и склоняет все больше профессиональных разработчиков писать свои программы на нем. На текущий момент трудно представить крупную компанию, не использующую его в своих проектах. Поэтому знание *TypeScript* будет весомым критерием при устройстве на работу своей мечты.

[00.2] Для чего нужен TypeScript

Поскольку сложность современных вэб приложений сопоставима с настольными, то прежде всего *TypeScript* предназначен для выявления ошибок на этапе компиляции, а не на этапе выполнения. Кроме того, за счет системы типов разработчики получают такие возможности, как подсказки и переходы по коду, которые значительно ускоряют процесс разработки. Помимо этого, система типов в значительной степени избавляет разработчиков от комментирования кода, которое отнимает значительное время. Также система типов помогает раньше выявлять проблемы архитектуры, исправление которых обходятся дешевле на ранних этапах.

Хотя на текущий момент практически невозможно найти библиотеку, которая бы не была портирована на *TypeScript*. *JavaScript* код, оставшийся от предыдущих проектов, не стоит списывать со счетов, поскольку его совместное использование не вызовет никакой проблемы. Компилятор *TypeScript* отлично справляется с динамическим *JavaScript* кодом, включенным в свою типизированную среду, и даже выявляет в нем ошибки. Кроме того, при компиляции *.ts* файлов в *.js* дополнительно генерируются файлы декларации *.d.ts*, с

помощью которых разработчики, создающие свои программы исключительно на *JavaScript*, получат все прелести типизированного автодополнения в любой современной среде разработки.

[00.3] Зачем разработчику TypeScript

TypeScript значительно сокращает время на устранение ошибок и выявление багов, которые порой не так просто определить в динамической среде *JavaScript*.

В случае, если для разработчика *TypeScript* является первым типизированным языком, следует знать, что его изучение значительно ускорит процесс профессионального роста, поскольку типизированный мир открывает аспекты программирования, которые не проявляются в языках с динамической типизацией.

Помимо этого, *TypeScript* позволяет писать более понятный и читаемый код, максимально описывающий предметную область, за счет чего архитектура приложения становится более выраженной, а разработка неявно увеличивает профессиональный уровень программиста.

Всё это в своей совокупности сокращает время разработки программы, снижая её стоимость и предоставляя разработчикам возможность поскорее приступить к реализации нового и ещё более интересного проекта.

Глава 01

Вступление

Тема типизации и всё то, что входит в её определение, является частой темой при обсуждении различных областей связанных с разработкой программ.

Любой профессионал который вовлечен в разработку, должен быть посвящен в тонкости типизации и уметь принимать решения в зависимости от выбранного языка программирования, что бы облегчить и ускорить написание кода и тем самым сократить затраты связанные с разработкой программного обеспечения.

Профессиональный разработчик обязан разбираться в типизации, для того, что бы уметь осмысленно использовать возможности того или иного языка программирования в очередном проекте.

Начинающему разработчику понимание типизации может помочь принять правильное решение, от которого будет зависеть вся его карьера. Ведь каждый язык программирования привносит в жизнь разработчика не только возможности, но и неповторимую идеологию.

Поэтому прежде чем приступить к изучению *TypeScript* стоит закрыть все вопросы связанные непосредственно с самой типизацией. Если данная тема кажется Вам не совсем понятной или вы вовсе с ней не знакомы, то данная глава поможет стать более компетентными в таких вопросах как

- Система типов
- Тип данных
- Типы значение
- Ссылочные типы
- Явные/неявные преобразования

- Типизация
- Типизированные/нетипизированные языки программирования
- Статическая/динамическая типизация
- Сильная/слабая типизация
- Явная/неявная типизация
- Вывод типов
- Совместимость типов

И перед тем как приступить к освещению указанных выше тем хотелось бы обратить внимание на то, что все примеры которые нуждаются в коде, будут продемонстрированы с применением языка *псевдо-TypeScript*. Псевдо означает, что могут быть использованы типы, которые в самом *TypeScript* не существуют.

Глава 02

Система типов, тип данных, значимые и ссылочные типы

Представьте, что вы находитесь в центре пустыни Невады и вокруг вас на максимальной скорости хаотично разъезжают сотни ревущих бульдозеров за рулем которых сидят макаки с завязанными глазами. Кажется, что дожить до момента когда у них закончится топливо практически не реально. Подобный пример максимально близко описывает отсутствие типизации. Противоположностью является продуманная дорожная инфраструктура состоящая не только из самих дорог, но и подземных\наземных переходов, светофоров и регулировщиков. Ключевым моментом является понимание, что ваша безопасность достигается путем *ограничения* транспорта. Другими словами типизация сопоставимая с дорожной инфраструктурой ограничивает возможности разработчика таким образом, что процесс создания программ становится для него более комфортным. Типизация не расширяет возможности разработчиков, она их ограничивает, пуская их энергию в правильное русло.

В зависимости от среды выполнения, операции над данными могут быть ограничены некими правилами. Для каждого конкретного вида данных декларируются конкретные правила. В роли конкретного свода правил выступает тип данных.

Среда выполнения, в которой отсутствует классификация данных и ограничения на операции над ними, не имеет типов, поэтому такую среду выполнения справедливо называть нетипизированной. И наоборот, среду выполнения, которая классифицирует данные и операции над ними, справедливо называть типизированной.

В основе любой типизированной среды выполнения лежит такое фундаментальное понятие, как *система типов*. Сложно сформулировать исчерпывающие определение термину *система типов*, затронув все аспекты, все значения, которые вкладывают в этот термин разработчики из различных областей, поэтому конкретизируем его самым простым определением.

[02.0] Система Типов

Система Типов — это совокупность правил назначающих конструкциям составляющим программу свойства (имеется ввиду характеристики) именуемые *типами*. Обычно к конструкциям нуждающимся в аннотации типов относятся переменные, поля и свойства объектов, а также параметры и возвращаемые функциями значения.

В основе системы типов любого языка программирования всегда лежит *базисная система типов* встроенных в язык. К базисным или встроенным типам относятся такие типы, как `byte`, `int`, `string`, `boolean`, `object` и им подобные. На их основе среда выполнения или разработчик могут определять типы данных более высокого уровня, например `Date` или `Array`.

[02.1] Тип данных (тип)

Понятие *тип* является фундаментальным в теории программирования. Тип данных (или просто тип) — это характеристика определяющая множество значений и операций, которые могут быть выполнены над этими данными.

В зависимости от языка программирования, тип данных может хранить информацию о данных, к которым относятся поля, свойства, методы и другие структуры языка, а также о том, в каком месте, в *стеке* (*stack*) или *куче* (*heap*) будет выделяться память во время выполнения программы и её объем; в каких операциях (как, например, сложение `+`, умножение `*`, присваивание `=` и т.д.) может участвовать тип данных.

Типы данных делятся на два вида:

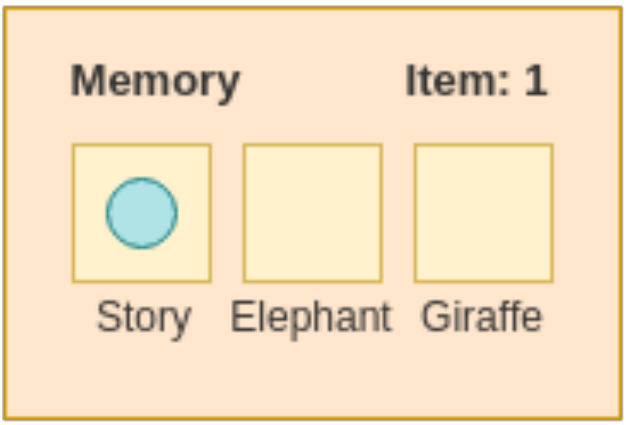
- *типы значения* (*value type*) — хранят значение (их ещё называют *значимыми типами*)
- *ссылочные типы* (*reference types*) — хранят ссылку на значение

При операции присваивания значения принадлежащего к значимому типу данные копируются (дублируются) в памяти. При операции присваивания значения, принадлежащему к ссылочным типам, копируется лишь ссылка на данные.

[02.2] Тип данных, передающийся по значению (примитивный тип)

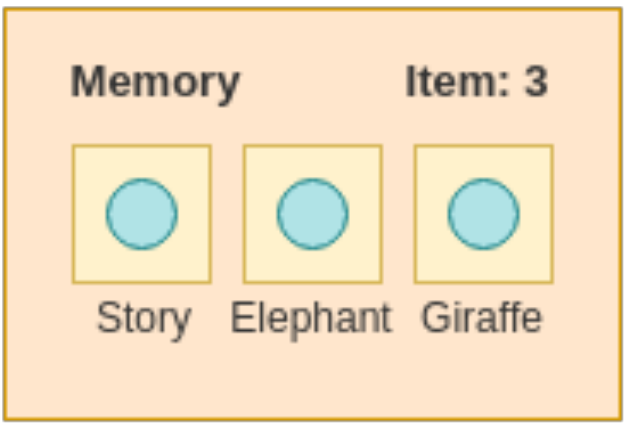
Когда переменная, ассоциированная со значением принадлежащим к значимому типу участвует в операции присвоения, операнд из левой части будет ассоциирован не со значением правого операнда, а с его копией. Другими словами, значение будет дублировано в памяти и переменные будут ассоциированы с разными значениями-участками памяти. При изменении любой переменной своего значения, значения других переменных затронуты не будут.

Передача по значению



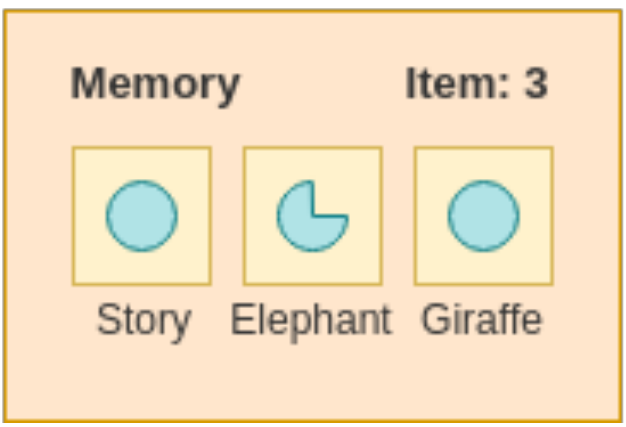
Step: 0

В хранилище Store хранится одно яблоко.



Step: 1

Даем яблоко слону и жирафу.



Step: 2

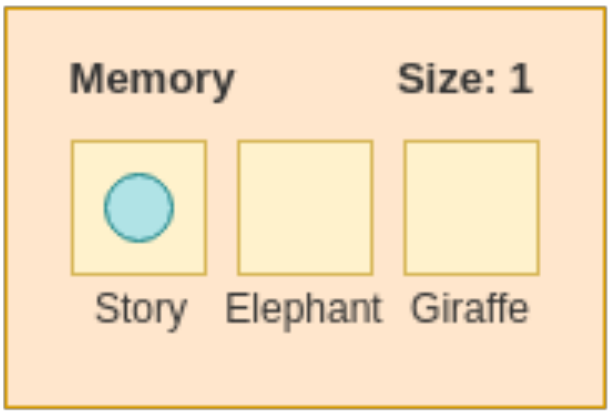
Слон откусывает часть яблока.

Обычно говорят, что переменные с типом значения хранят значение и передаются по значению.

[02.3] Тип данных, передающийся по ссылке

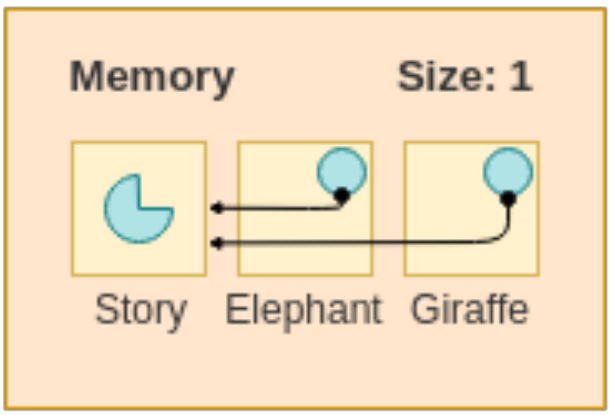
Если переменная ассоциированная со ссылочным типом данных участвует в операции присваивания, операнду из левой части будет присвоена ссылка на значение, с которым ассоциирован операнд из правой части. Другими словами обе переменные будут ассоциированы с одним и тем же значением и участком памяти. В таком случае, если изменить значение любой ссылочной переменной, изменения затронут все переменные, ассоциированные с этим значением.

Передача по ссылке



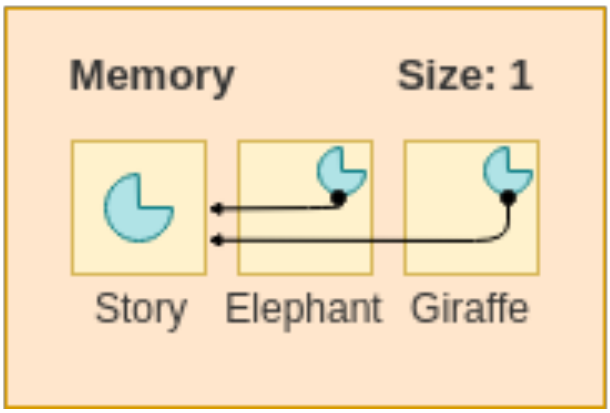
Step: 0

В хранилище Store хранится одно яблоко.



Step: 1

Даем яблоко слону и жирафу.



Step: 2

Слон откусывает часть яблока.

Обычно говорят, что переменные ссылочного типа ссылаются на значение и передаются по ссылке.

Глава 03

Связывание, типизация, вывод типов

Данная глава поможет разобраться как именно компилятор определяет принадлежность элементов кода к конкретным типам, что именно он подразумевает под термином типизация и какую неоценимую пользу приносит вывод типов.

[03.0] Обработка кода компилятором

Прежде чем код высокого уровня превратится в последовательность машинных инструкций, которые смогут выполниться на компьютере, его нужно к этому подготовить. Детали зависят от конкретного языка программирования, но общим для всех является то, что компилятор или интерпретатор должен построить понятную ему модель на основе определяемого языком синтаксиса. Построение модели разделяется на этапы *лексического*, *синтаксического* и *семантического* анализа.

[03.1] Лексический анализ (токенизация - tokenizing)

На этапе *лексического анализа* исходный код разбивается на лексемы, при этом пробелы и комментарии удаляются.

Лексема — это последовательность допустимых символов.

К примеру, рассмотрим следующий код:

ts

```ts
var n = 100 + 2 * 3;
```

Этот код будет разбит на девять лексем: `var`, `n`, `=`, `100`, `+`, `2`, `*`, `3`, `;`.

[03.2] Синтаксический анализ (разбор - parsing)

На этапе *синтаксического анализа* обычно строится дерево разбора путем определения последовательности лексем. Его ещё называют *синтаксическим деревом*.

В качестве примера можно привести предыдущий код, в котором порядок операций вычисления выражений изменен из-за присутствия в нем оператора умножения. Простыми словами результат выражения будет вычисляться не слева направо, а справа налево. Именно на этапе синтаксического анализа и определяется правильная последовательность.

ts

```ts
var n = 100 + 2 * 3;
```

[03.3] Семантический анализ

На этапе *семантического анализа*, который считается самым важным этапом, устанавливается семантика (смысл) построенного на предыдущем шаге дерева разбора.

В процессе семантического анализа в зависимости от среды выполнения, может осуществляться привязка идентификаторов к типу данных, проверка совместимости, а так же вычисление типов выражений.

Результатом выполнения этапа является расширенное дерево разбора, новое дерево, либо другие структуры, которые будут способствовать дальнейшему разбору и преобразованию промежуточного кода.

Независимо от того, какая структура будет получена на этапе семантического анализа, такие конструкции как переменные, поля, свойства, параметры и возвращаемые функциями и методами значения установят связь с типами данных, к которым они принадлежат. Другими словами, произойдет связывание.

[03.4] Связывание (Binding)

Связывание (binding) — это операция ассоциирования конструкций нуждающихся в аннотации типа с типами данных к которым они принадлежат.

[03.5] Типизация

Типизация — это процесс установления принадлежности результата выражения к типу данных с последующим связыванием его с объектом данных, представляющим его в программе.

[03.6] Вывод Типов (type inference)

Вывод типов — это возможность компилятора (интерпретатора) определять тип данных, анализируя выражение.

Для переменной не имеющей явной аннотации типа, компилятор выведет тип данных на основе результата присваиваемого выражения.

ts

```ts
// так разработчик видит
var n = 5 + 5; // так видит компилятор var n: number = 5 + 5;
```

В большинстве случаев термин *вывод типов* употребляется относительно языков нуждающихся в предварительной компиляции на этапе которой и происходит установление типов данных путем их вывода. В контексте интерпретируемых языков вывод типа упоминается реже - в них при помощи вывода типов производятся оптимизации на этапе выполнения.

Глава 04

Преобразование типов

Типы относящиеся к одной категории так же как и различную валюту можно конвертировать друг в друга. И точно также при этом могут возникнуть не состыковки. Представьте, что вам необходимо обменять евро на доллары которые стоят в два раза дешевле. При этом они существуют только в виде купюр номиналом в один доллар. При таких условиях за одно евро вы получите два доллара. В случае роста курса доллара обмен без остатка стал бы невозможен. Если бы евро стоило полтора доллара, то при обмене вы бы получили только один доллар, так как номинала в половину доллара не существует. Другими словами вы бы потеряли часть своих доходов. Точно также и с типами при конвертации которых может происходить частичная потеря данных.

Преобразование типов — очень важная часть типизированного мира.

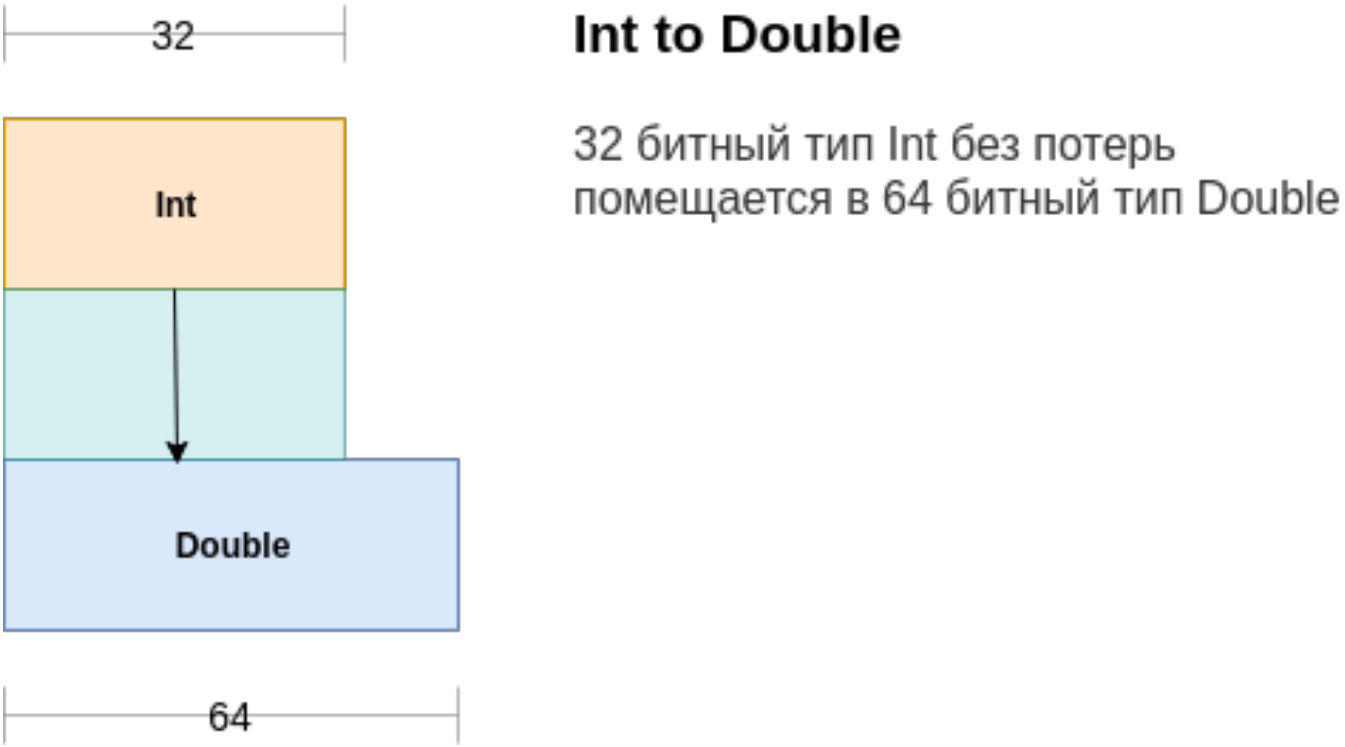
Преобразование типов (type conversion, typecasting) — это процесс заключающийся в преобразовании (конвертации) значения одного типа в значение другого типа. Преобразование очень важная часть типизированного мира, работать с которой приходится постоянно. Поэтому понимать тонкости данного процесса очень важно.

Преобразование типов делятся на *явные* и *неявные* преобразования.

[04.0] Неявные Преобразования

Неявные преобразования не требуют никаких языковых конструкций и не приводят к утрате данных. Это как если бы в магазине принимали и доллары и евро, что избавило бы покупателей от процесса обмена.

Примером неявного преобразования может служить преобразование значения типа `int` в тип `double`. Являясь 64-битным вещественным типом, `double` может хранить любое значение 32-битного целочисленного типа `int`.



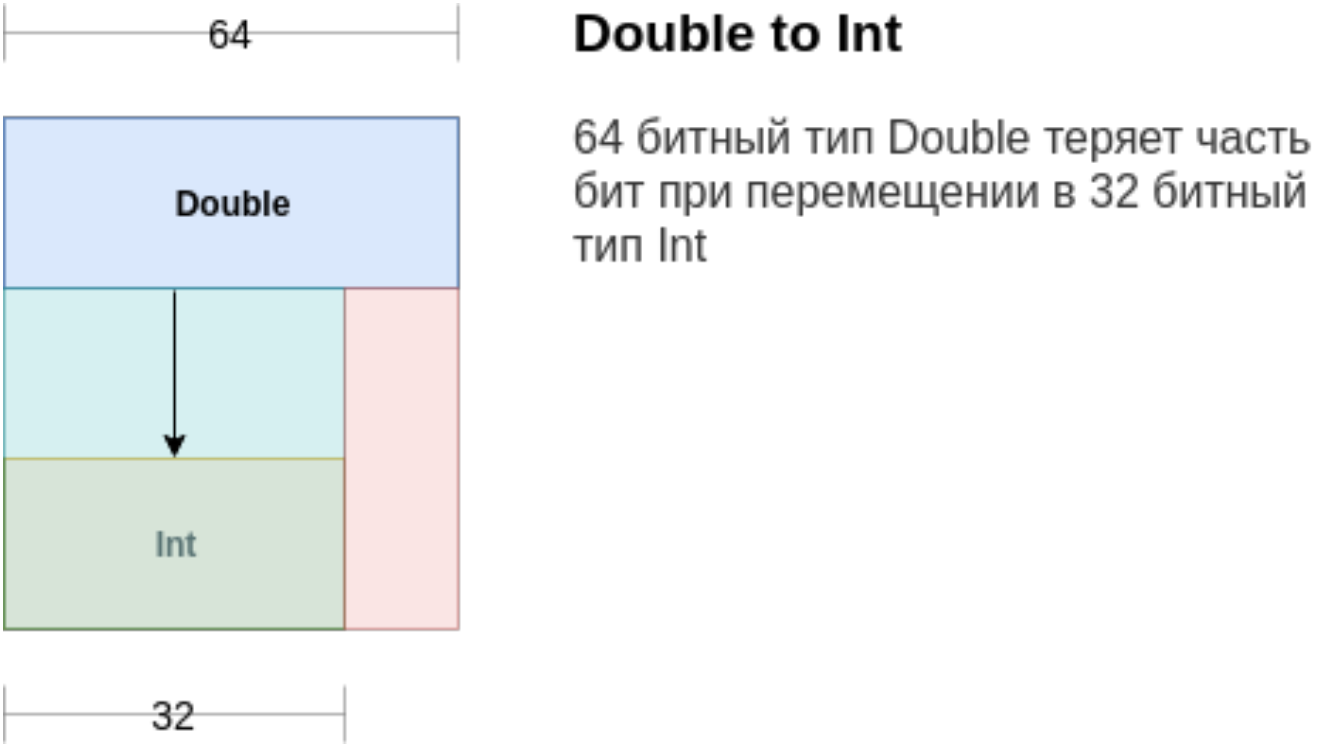
Как показано на изображении выше, преобразование `int` в `double` не приводит к потере данных, так как 32-битное значение умещается в 64 битах.

[04.1] Явные Преобразования

Явные преобразования, которые для ясности часто называют *приведением типов*, происходят при участии разработчика и требуют указания языковых конструкций, называемых *операторами приведения*. Приведение типов требуется тогда, когда в процессе преобразования данные могут быть утрачены или по каким-то причинам процесс может завершиться ошибкой. По аналогии из жизни данный процесс сопоставим

с ранее приведенным примером в котором евро стоило полтора доллара, то есть обмен без потери не возможен.

Самым простым примером явного преобразования служит преобразование типа `double`, используемого для хранения 64-битных значений с плавающей запятой, к целочисленному типу `int`.



Изображение выше демонстрирует потерю данных при преобразовании 64-битного типа `double` к 32-битному типу `int`.

Глава 05

Типизированные и нетипизированные языки программирования

Языки программирования по признакам типизации делятся на две категории — *нетипизированные* (untyped) и *типизированные* (typed). К нетипизированным можно отнести например *Assembler* или *Forth*, а к типизированным *Java*, *C#*, *JavaScript* или *Python*.

[05.0] Нетипизированные языки

Нетипизированные или *бестиповые* языки, как правило, очень старые и низкоуровневые языки на которых программы являются набором машинных команд и пишутся, в основном, для взаимодействия с аппаратным обеспечением (с железом). Бестиповые языки позволяют производить любые операции над любыми данными, которые представлены в них как цепочки бит произвольной длины.

[05.1] Типизированные языки

Типизированные языки сделали разработку программ более осмысленной и эффективной. Этот факт оказался настолько весомым, что в основу всех современных языков легло такое фундаментальное понятие, как система типов. Поскольку каждый новый типизированный язык программирования создавался для решения особых задач, все они стали различаться по видам типизации, которая делится на:

- *статически* и *динамически* типизированные
- *сильно* и *слабо* типизированные
- *явно* и *неявно* типизированные

Глава 06

Статическая и динамическая типизация

При таких операциях, как сложение целых чисел, машина не способна проверить принадлежность к типу переданных ей операндов. Для неё это просто последовательность битов. Отсутствие механизма проверки типов делает низкоуровневые программы ненадежными.

Контроль типов устанавливает принадлежность каждого операнда к конкретному типу данных, а также проверяет его на совместимость с оператором, участвующим в текущей операции. Любое несоответствие приводит к возникновению ошибки.

Этап на котором происходит контроль типов, определяет вид типизации. Контроль типов который происходит на этапе компиляции, называется *статическим контролем типов* или *статической типизацией*. Контроль типов, который происходит на этапе выполнения программы, называется *динамическим контролем типов* или *динамической типизацией*.

[06.0] Статическая типизация (Static Typing)

Статическая типизация обуславливается тем, что нуждающаяся в аннотации типа конструкция связывается с типом данных в момент объявления на этапе компиляции. При этом связь с типом данных остается неизменна.

Другими словами, до того момента, как программа будет запущена, компилятор осуществляет проверку совместимости типов участвующих в различных операциях.

Сущность, представляющая любую конструкцию нуждающуюся в аннотации типа, связывается с типом данных при объявлении. При этом связь с типом данных в дальнейшем не может быть изменена.

За счет того, что большая часть проверок происходит на этапе компиляции, программа обладает большей производительностью из-за отсутствия проверок во время выполнения.

Языки со статической проверкой типов значительно повышают семантическую привлекательность кода делая его более читабельным.

Благодаря статическому контролю типов, редактор кода способен на основе синтаксического анализа выводить вспомогательную информацию, ускоряющую разработку.

К статически типизированным относятся такие языки, как *C#*, *Java*.

[06.1] Динамическая Типизация (Dynamic Typing)

Динамическая типизация обусловлена тем, что конструкция нуждающаяся в аннотации типа, связывается с типом данных на этапе выполнения программы в момент присвоения значения. При этом связь с типом данных может быть изменена.

В динамической среде выполнения тип данных может быть определен только на основании вычисленного результата выражения. При этом операция присвоения значения может изменить тип данных сущности с которым она была связана ранее.

Динамический контроль типов проверяет операнды на совместимость типов и совместимость с оператором в каждой отдельной операции. Проверка типов данных во время выполнения программы отрицательно сказывается на её производительности. Поэтому языки с динамическим контролем типов проигрывают в производительности языкам со статическим контролем типов у которых совместимость проверяется на этапе компиляции.

По причине выявления несовместимости типов только во время выполнения программы ошибки выявляются менее эффективно. Программа не узнает о скрывающейся в ней ошибке, пока не дойдет очередь выполнения проблемного участка кода.

К тому же, редакторы кода, даже современные, испытывают трудности при синтаксическом анализе кода необходимого для вывода информации облегчающей разработку.

К динамически типизированным языкам относятся такие языки как *PHP*, *Python*, *JavaScript*.

И напоследок остается добавить, что в языках со статической типизацией существуют возможности динамического связывания.

Глава 07

Сильная и слабая типизация

Типизированные языки программирования также разделяются на *сильно* и *слабо* типизированные. Очень часто можно услышать их неверные синонимы *строгая* и *нестрогая* типизация. Эти синонимы считаются некорректными так как произошли в результате неверного перевода англоязычных терминов *strongly typed* и *weakly typed*, что дословно переводится как *сильно типизированные* и *слабо типизированные*.

[07.0] Сильная типизация (strongly typed)

Языки с *сильной типизацией* не разрешают выполнение выражений с несовместимыми типами и не выполняют неявное преобразование типов в ситуациях, когда нужно выполнять преобразование явно.

Сильно типизированный язык не позволит такие операции, как умножение числа на массив и не выполнит неявного преобразования объекта к строке.

ts

```ts
var multi = (a, b) => a * b;

var number = 5;
var array = [0, 1, 2];

var result = multi(number, array[2]); // Ок, вернёт 10
var result = multi(number, array); // Ошибка
```

```
var welcome = name => 'Hello' + ' ' + name;

var user = {
    name: 'Bob'
};

var result = welcome(user.name); // Ок, вернёт 'Hello Bob'
var result = welcome(user); // Ошибка
```

В языках с сильной типизацией при операциях способных привести к потере или порче значения возникает ошибка. Благодаря этому сильно типизированные языки в меньшей степени подвержены багам.

К языкам с сильной типизацией можно отнести такие языки, как *Java*, *C#* и другие.

[07.1] Слабая типизация (weakly typed)

Языки со *слабой типизацией* разрешают выполнение выражений с любыми типами и самостоятельно выполняют неявное преобразование.

Например, в *JavaScript* при операции сложения строки и числа второй операнд может быть неявно приведён к строке, а при сложении строки и объекта объект будет неявно приведён к значению возвращаемому объектом из метода `toString()`, то есть строке.

ts

```
var add = (a, b) => a + b;

var results = add(5, 5); // Ок, вернёт 10
var result = add('Hello', 5); // Ошибки не будет, но результат будет
Hello5

var welcome = name => 'Hello' + ' ' + name;

var user = {
    name: 'Bob'
};

var result = welcome(user.name); // Ок, вернёт 'Hello Bob'
var result = welcome(user); // Ошибки не будет, но в результате вернёт
'Hello [object Object]'
```

Из-за того, что в языках со слабой типизацией допускаются неявные преобразования существует высокая вероятность возникновения трудно выявляемых багов. Причиной появления подобных багов является отсутствие возникновения ошибок в некоторых операциях неявного преобразования. Программа получает данные ожидаемого типа, но непредсказуемого значения.

ts

```
var valueA = 5 + '5'; // Вернёт 55 (string) вместо 10 (number)
var valueB = valueA * 2; // Вернет 110 (number) вместо 20 (number)
```

К языкам со слабой типизацией относятся *JavaScript* или *PHP**.

Глава 08

Явная и неявная типизация

Помимо разделения на *статическую/динамическую* и *сильную/слабую* типизацию, типизированные языки программирования разделяются по признакам *явной/неявной* типизации.

[08.0] Явная типизация (explicit typing)

Язык с *явной типизацией* предполагает, что указание принадлежности языковых элементов к конкретному типу возлагается на разработчика.

Явная типизация делает код более понятным (читабельным) и позволяет разработчикам не знакомым с ним, быстрее включатся в процесс разработки, тем самым снижая время на модификацию программы.

Явная типизация обязывая явно указывать типы способствует развитию у разработчика навыков необходимых для проектирования архитектуры программ.

В львиной доле языков с явной типизацией существует возможность указывать типы неявно.

Рассмотрим пример кода демонстрирующего явную типизацию.

ts

```
class Controller {
    public check(eggs: IEgg[]): boolean {
```

```
        const isValid: boolean = Validator.valid(eggs);

        return isValid;
    }
}
```

Класс `Controller` содержит метод `check`, который имеет один обязательный параметр `eggs` с типом `IEgg[ ]` и возвращающий тип `boolean`. Если бы не явно указанные типы параметров, то разработчикам пришлось бы только гадать с чем именно им предстоит работать. Это же относится и к возвращаемому типу.

Результатом выполнения `Validator.valid(eggs)` внутри метода `check` является возвращаемое из функции значение типа `boolean`. Если бы при объявлении переменной `isValid` тип `boolean` не был бы указан явно, то было бы сильно сложнее догадаться, что же возвращает метод `valid`.

Разработчику впервые увидевшему этот код или тому, кто имел с ним дело очень давно, не составит труда разобраться за, что отвечает данный участок кода.

К языкам с явной типизацией относятся *C++*, *C#* и многие другие.

[08.1] Неявная типизация (implicit typing)

Язык с *неявной типизацией* при объявлении языковых элементов не требует от разработчика указания принадлежности к конкретному типу данных и возлагает определение типов на компилятор или интерпретатор.

За основу примера неявной типизации возьмем код из предыдущего примера, только лишим его признаков характерных для явной типизации.

ts

```
class Controller {
    check(eggs) {
        const isValid = Validator.valid(eggs);

        return isValid;
    }
}
```

Этот код стал занимать меньше места, что является одним из нескольких доводов, которые можно услышать в пользу языков с неявной типизацией. Но на самом деле это не так.

На практике считается хорошим тоном при объявлении языковых элементов уделять особое внимание именованию. Ведь именно от выбора названия будет зависеть, то время, которое уйдет у программиста на понимание участка кода при отладке, рефакторинге или модернизации.

Те же рассуждения, в процессе которых происходит рождение более информационного названия, приводят к более детальному осмыслению кода.

ts

```ts
class EggController {
   checkEgg(eggAll) {
       const isEggValid = EggValidator.qualityEggValid(eggAll);

       return isEggValid;
   }
}
```

Именно по этой причине правило именования распространяется и на языки с явной типизацией.

А тот факт, что неявная типизация позволяет реализовывать несложные алгоритмы с меньшими временными затратами, разбивается о возможность всех современных языков с явной типизацией указывать тип неявно, что достигается благодаря выводу типов.

ts

```ts
class EggController {
   public checkEgg(eggAll: IEgg[]): boolean {
       const isEggValid: boolean = EggValidator.qualityEggValid(eggAll);

       return isEggValid;
   }
}
```

К языкам с неявной типизацией относятся такие языки, как *JavaScript*, *PHP* и другие.

Глава 09

Совместимость типов на основе вида типизации

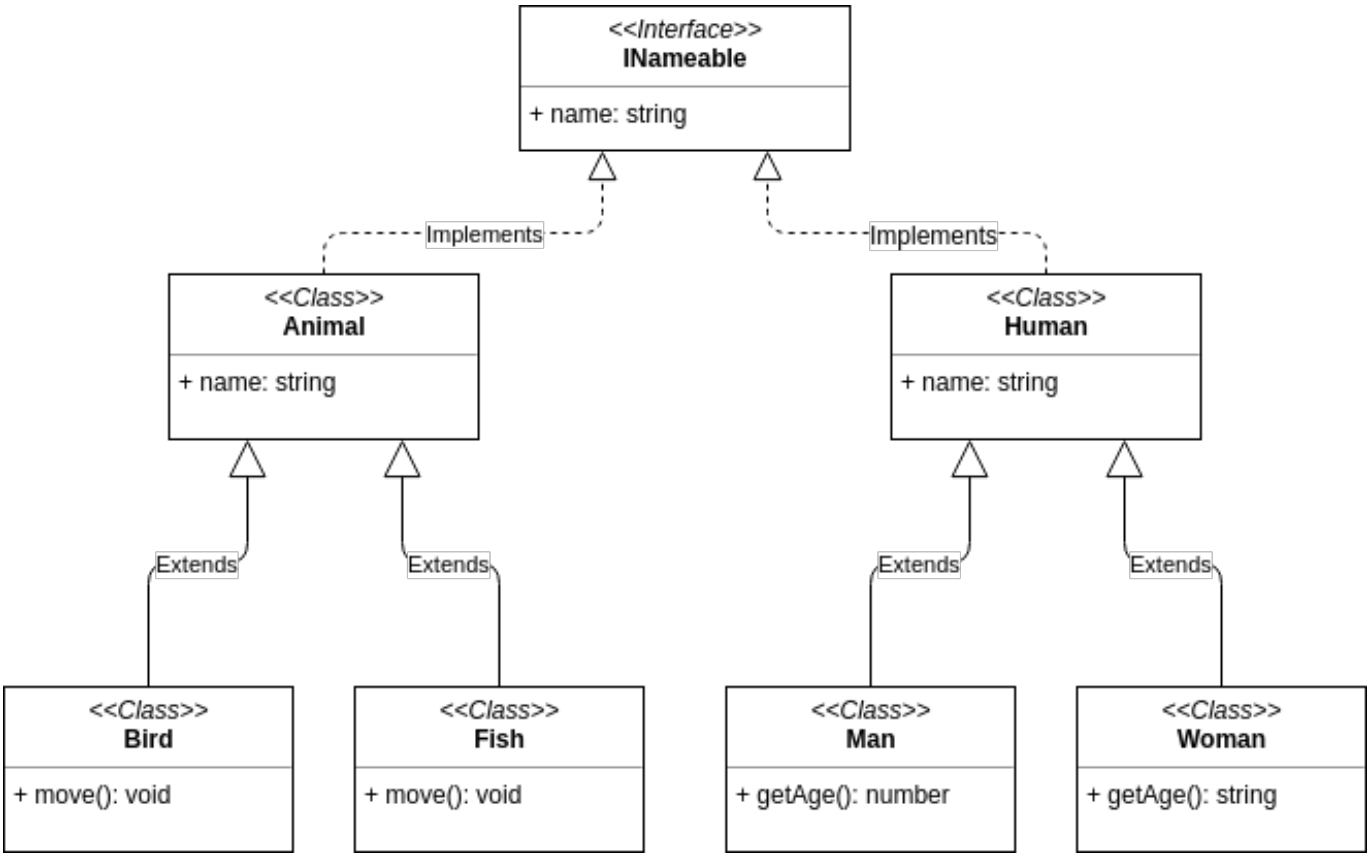
Каждый раз при присваивании значения компилятор подвергает его проверке на совместимость с ожидаемым типом. Компилятор словно секюрити стоящий на фейсконтроле ночного клуба решает пропускать или нет то, или иное значение. Подобные решения принимаются на основании правил, которые зависят не только клуба (языка программирования), но также вида проводимых мероприятий. Понять логику на которую опирается компилятор при определении совместимости нам поможет текущая и следующая глава. Но сначала давайте выведем правило самой совместимости.

[09.0] Совместимость типов (Types Compatibility)

Совместимость типов — это совокупность правил, на основе которых программа, анализируя два типа, принимает решение о возможности одного типа заменить другой таким образом, что бы замена не нарушила выполнение программы. Простыми словами, совместимость типов — это механизм, по которому происходит сопоставление типов.

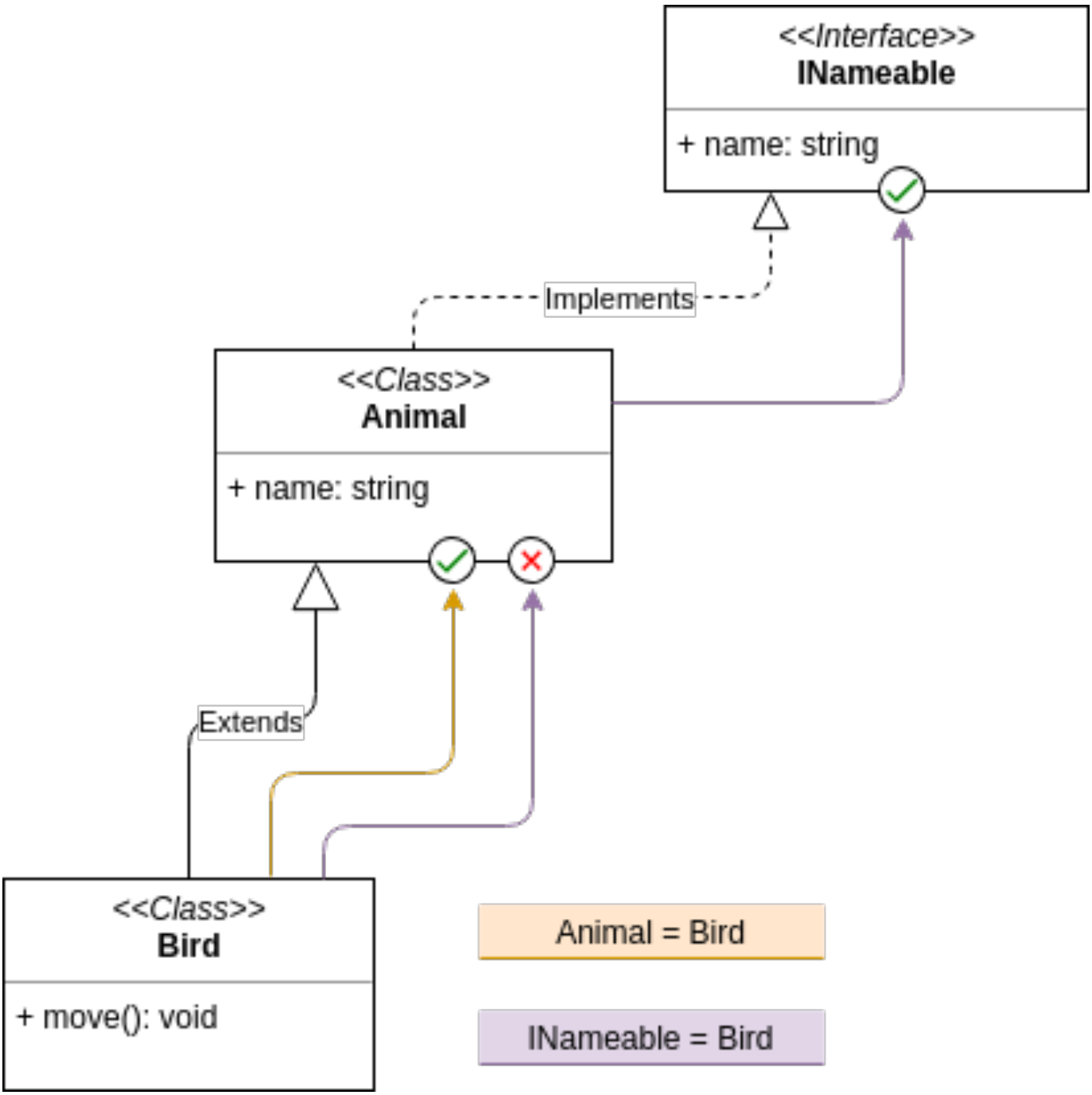
Существует несколько основных механизмов и выбор конкретного зависит от случая, при котором возникает потребность сопоставления типов данных. Один из таких механизмов состоит из совокупности правил составляющих такое понятие как типизация. Из существующего множества правил можно выделить несколько групп, которые образуют три вида типизации — *номинативную*, *структурную* и *утиную*.

Чтобы различия между ними были более очевидными, все они будут рассматриваться на одном примере, диаграмма которого показана ниже.

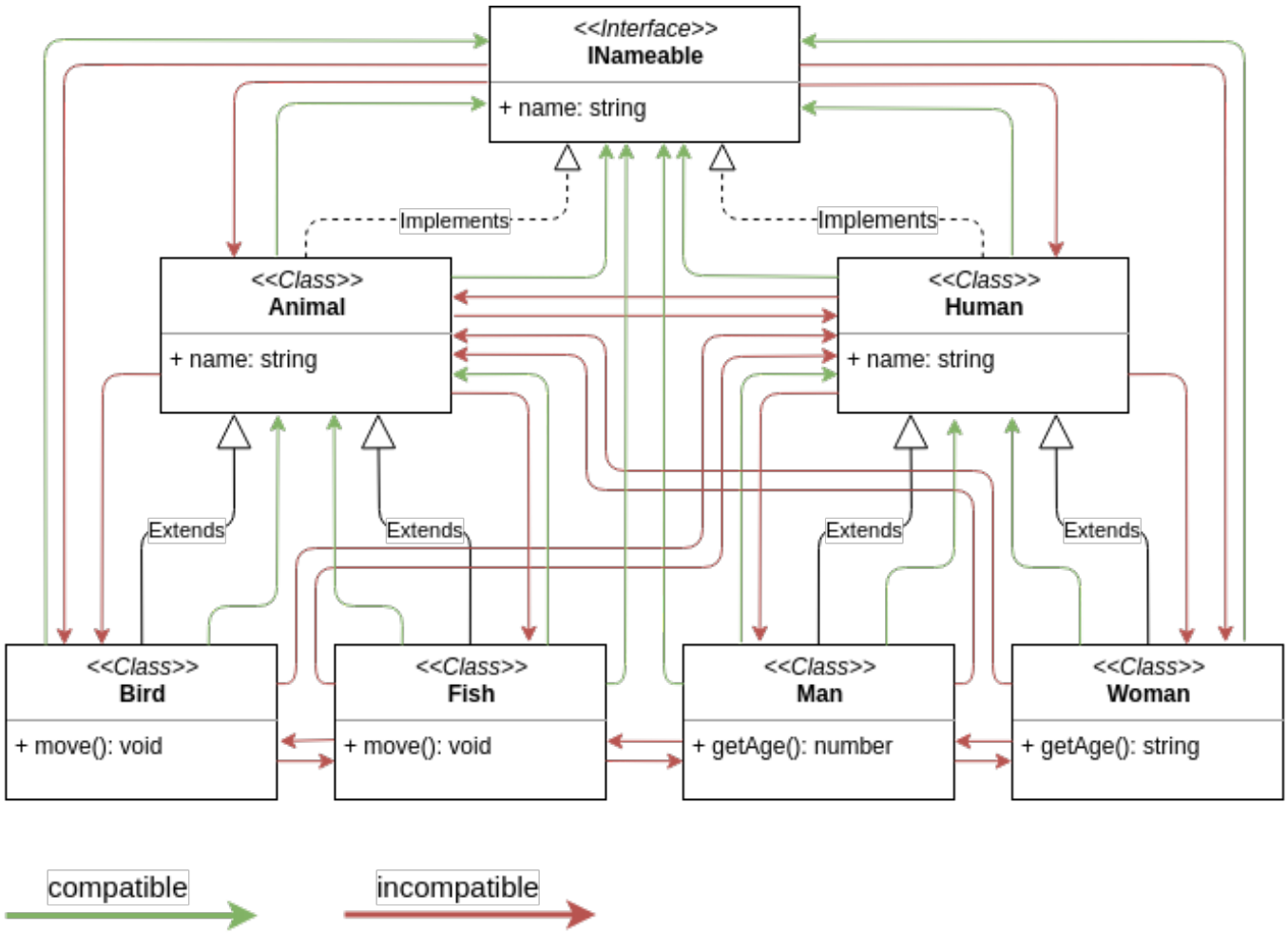


[09.1] Номинативная Типизация (nominative typing)

Номинативная типизация (nominative typing) устанавливает совместимость типов основываясь на идентификаторах типов (ссылках). Простыми словами, при проверке на совместимость компилятор проверяет иерархию типов на признаки наследования и реализацию интерфейсов. То есть, тип `B` будет совместим с типом `A` только тогда, когда он является его предком (`extends`). Кроме того, тип `B` будет совместим с интерфейсом `IA` только в том случае, если он или один из его предков реализует его явно (`implements`).



Как можно понять по изображению выше, при проверке на совместимость типа `Bird` с типом `Animal` компилятор обходит дерево в поисках наличия ссылки на тип `Animal` и обнаружив её, приходит к выводу, что типы совместимы. Тот же самый процесс требуется для установления совместимости типа `Bird` с типом интерфейса `INameable`. Полная картина совместимости изображена на диаграмме ниже.

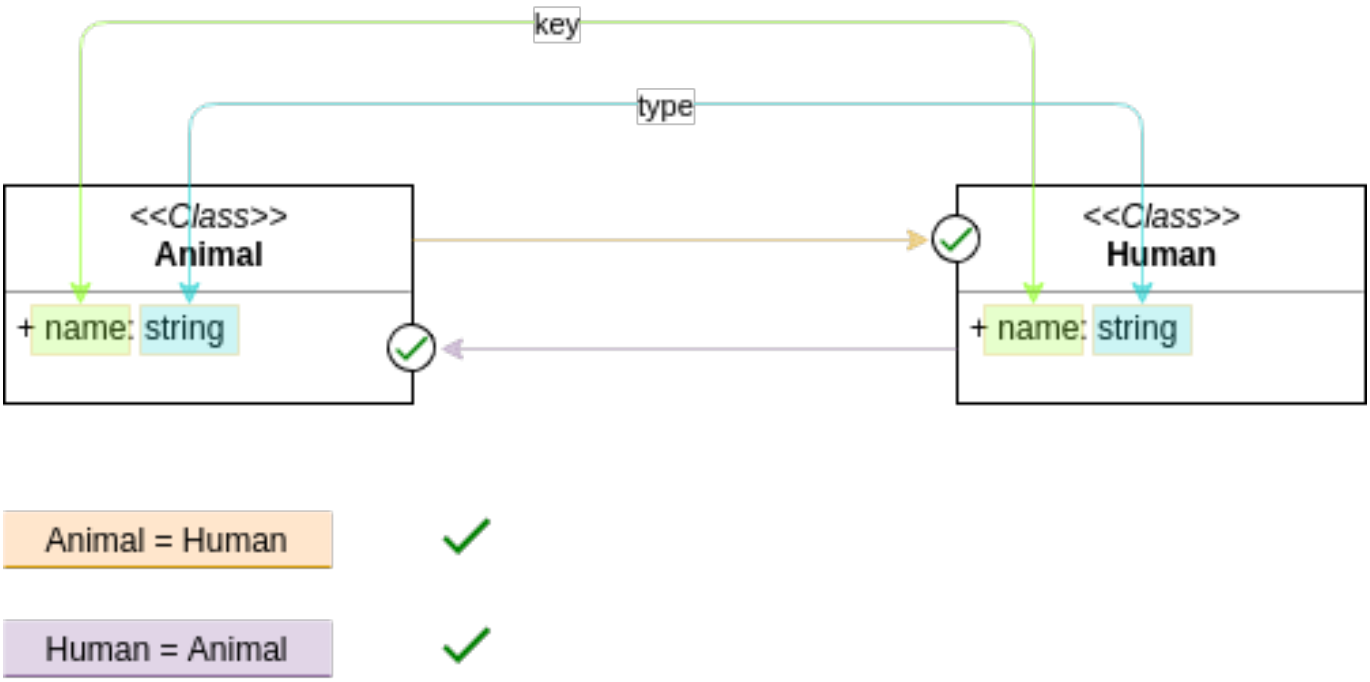


Номинативная типизация присуща исключительно статически типизированным языкам.

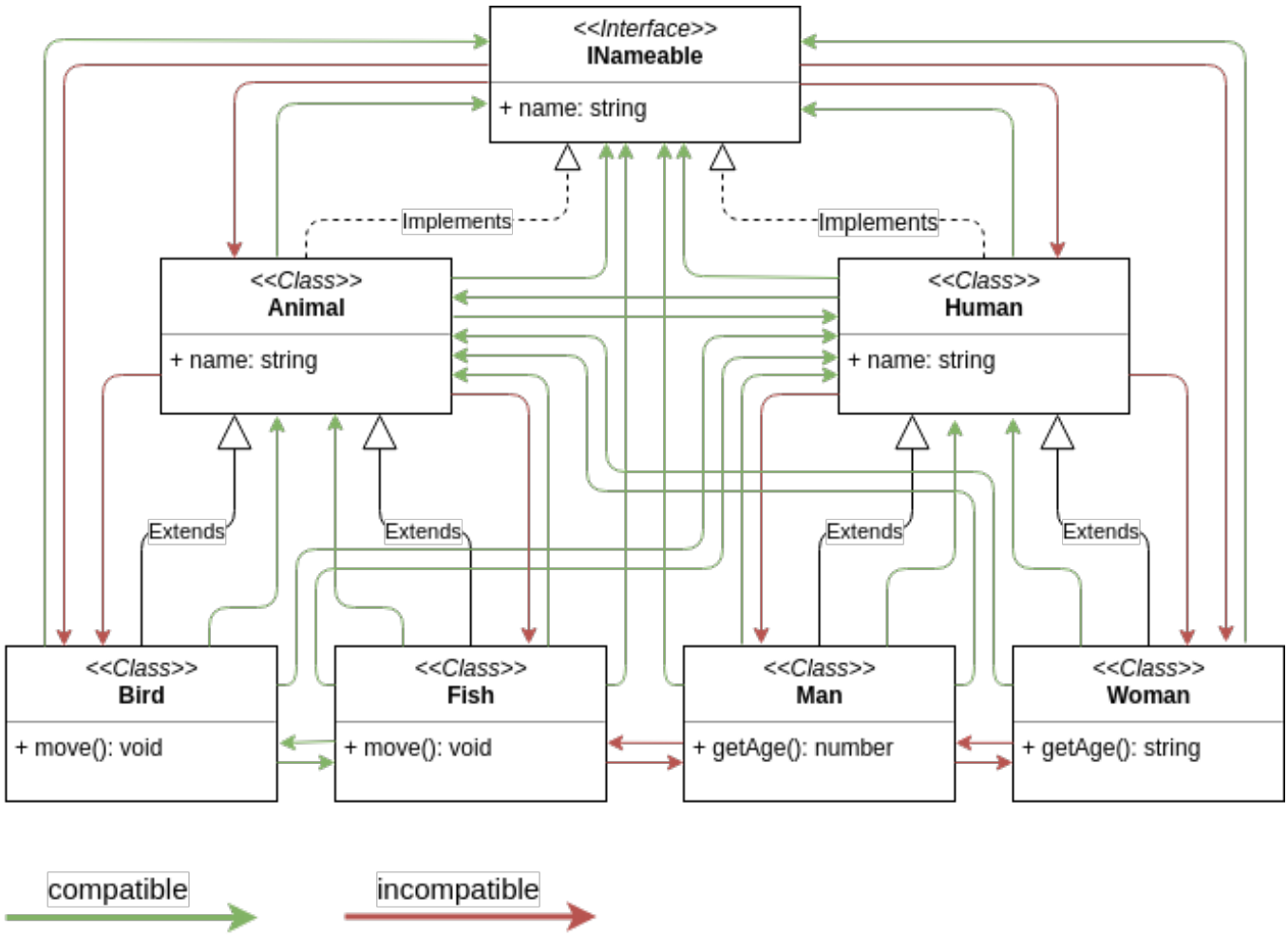
К языкам с номинативной типизацией относятся *Java*, *C#* и другие.

[09.2] Структурная Типизация (structural typing)

Структурная типизация — это принцип, определяющий совместимость типов основываясь не на иерархии наследования или явной реализации интерфейсов, а на их описании.



Компилятор считает типы совместимыми, если сопоставляемый тип имеет все признаки типа, с которым сопоставляется. Чтобы быть совместимым, сопоставляемый тип должен иметь те же ключи с теми же (идентичными или совместимыми) типами, что и тип, с которым происходит сопоставление. Полная картина совместимости в структурной типизации изображена на диаграмме ниже.



Структурная типизация присуща исключительно языкам с явной типизацией (глава [“Экскурс в типизацию - Сильная и слабая типизация”]).

К языкам со структурной типизацией относятся *TypeScript*, *Scala* и им подобные.

[09.3] Утиная Типизация (Duck typing)

Утиная типизация, как и в случае со структурной типизацией — это принцип, определяющий совместимость типов основываясь не на иерархии наследования или явной реализации интерфейсов, а на их описании. Утиная типизация ничем не отличается от структурной, за исключением того, что присуща лишь языкам с *динамическим связыванием* (динамическая типизация).

Термин «Утиная типизация» произошёл от английского выражения *duck test*, который в оригинале звучит как: *"Если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка"*.

Так как утиная типизация не отличается от структурной, то в качестве примеров совместимости можно воспользоваться диаграммой из предыдущего раздела, посвященного структурной типизации.

К языкам с утиной типизацией относятся *Python*, *JavaScript* и другие.

Глава 10

Совместимость типов на основе вариантности

Помимо того, что совместимость типов зависит от вида типизации, которая была подробно разобрана в главе “Экскурс в типизацию - Совместимость типов на основе вида типизации”, она также может зависеть от такого механизма как вариантность.

[10.0] Вариантность

Вариантность — это механизм переноса иерархии наследования типов на производные от них типы. В данном случае производные не означает *связанные отношением наследования*. Производные, скорее, означает *определяемые теми типами, с которых переносится наследование*.

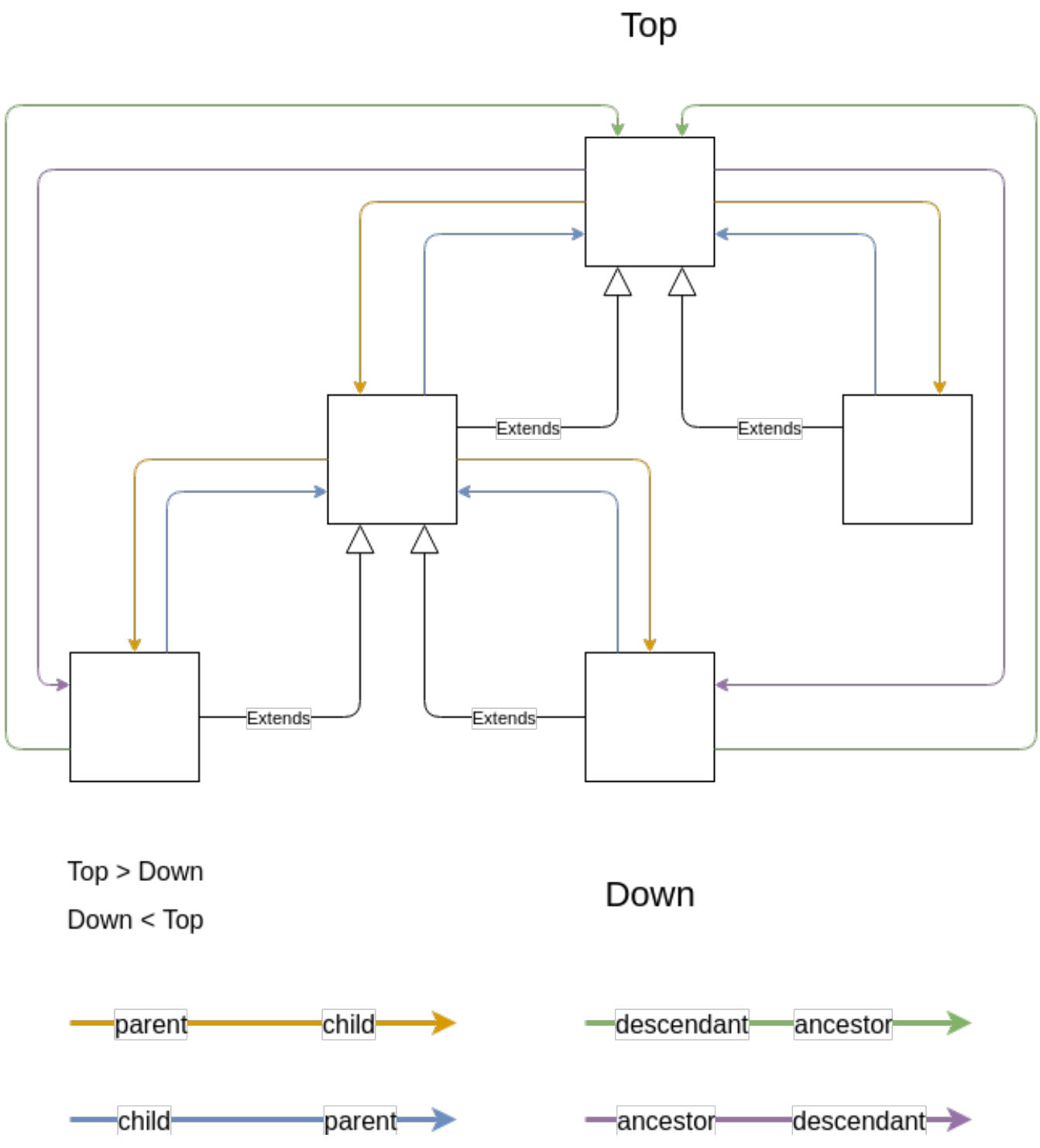
Если вы впервые сталкиваетесь с этим понятием и определение вариантности кажется бессмысленным набором слов, то не стоит расстраиваться, эта тема очень простая, в чем вы сами скоро убедитесь.

В основе системы типов могут быть заложены следующие виды вариантности — *ковариантность*, *контравариантность*, *инвариантность* и *бивариантность*. Кроме того, что система типов использует механизм вариантности для своих служебных целей, она также может предоставлять разработчикам возможность управлять им при помощи способов зависящих от конкретного языка.

Но прежде чем познакомиться с каждым из этих видов вариантности отдельно, стоит сделать некоторые уточнения касательно иерархии наследования.

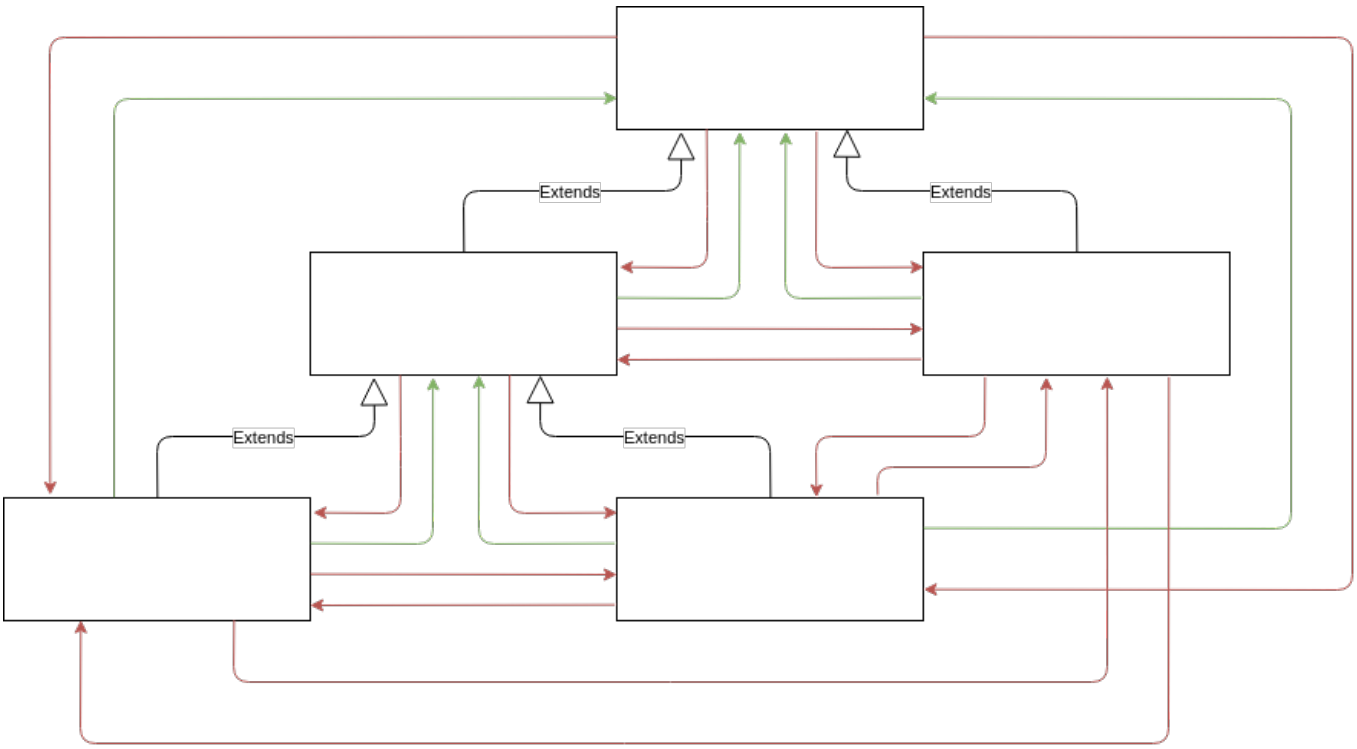
[10.1] Иерархия наследования

Иерархия наследования — это дерево, с расположенным вверху *корнем* или самый *базовым типом* (*менее конкретный тип*), ниже которого располагаются его *подтипы* (*более конкретные типы*). В случаях преобразования подтипа к базовому типу говорят, что выполняется *восходящее преобразование* (*upcasting*). И наоборот, когда выполняется приведение базового типа к его подтипу, говорят, что выполняется *нисходящее приведение* (*downcasting*). Отношения между супертипом и его подтипом описываются как отношение *родитель-ребенок* (parent-child). Отношения между родителем типа и его ребенком описываются как *предок-потомок* (ancestor-descendant). Кроме того, при логическом сравнении тип находящийся выше по дереву, больше (`>`) чем тип находящийся ниже по дереву (и наоборот). Можно сказать, что `parent > child` , `child < parent` , `ancestor > descendant` , `descendant < ancestor` . Все это представлено на диаграмме ниже.

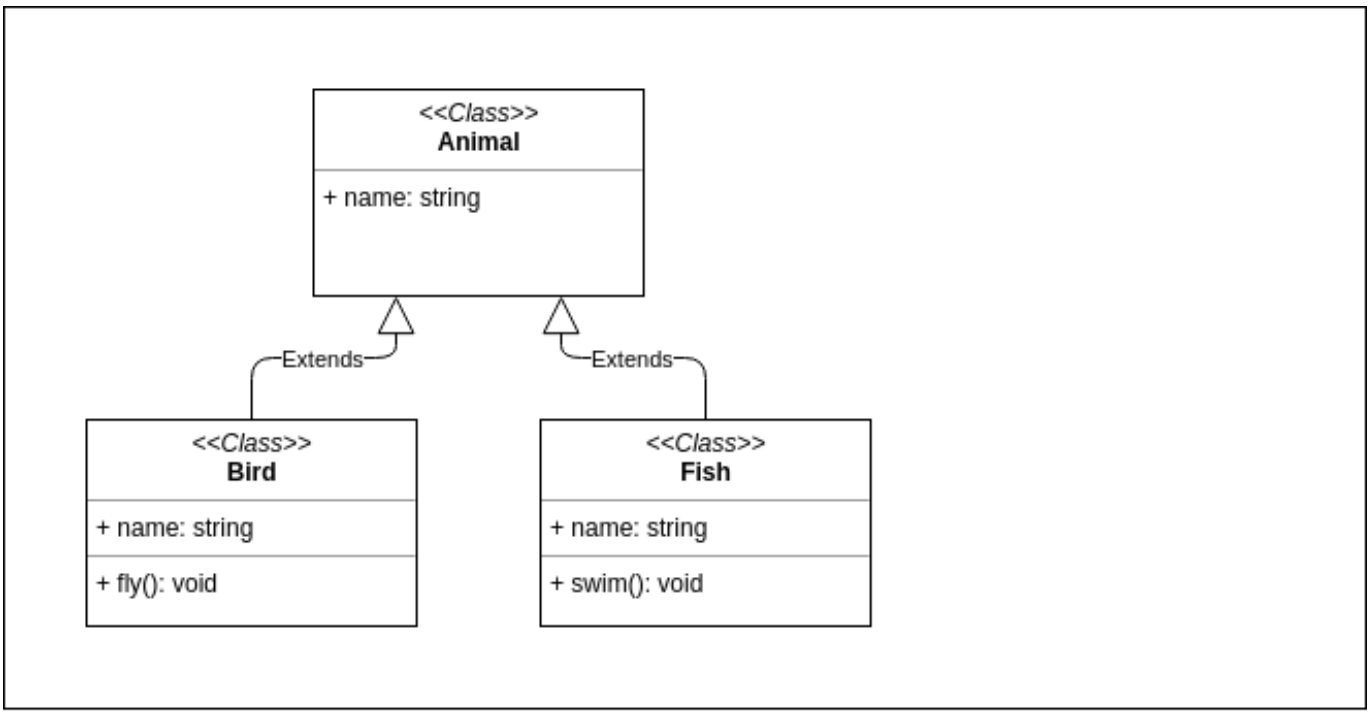

Top
Extends
Extends
Extends
Extends
Top > Down
Down < Top
Down
parent
child
descendant
ancestor
child
parent
ancestor
descendant

[10.2] Ковариантность

Ковариантность — это механизм позволяющий использовать более конкретный тип там, где изначально предполагалось использовать менее конкретный тип. Простыми словами, совместимыми считаются типы, имеющие отношение `A > B` и `A = B`.



Ковариантность не рекомендуется в местах, допускающих запись. Чтобы понять смысл этих слов, ниже представлена диаграмма, которая иллюстрирует, как через базовый тип можно добавить в массив с подтипом другой, несовместимый подтип и тем самым нарушить типобезопасность программы.



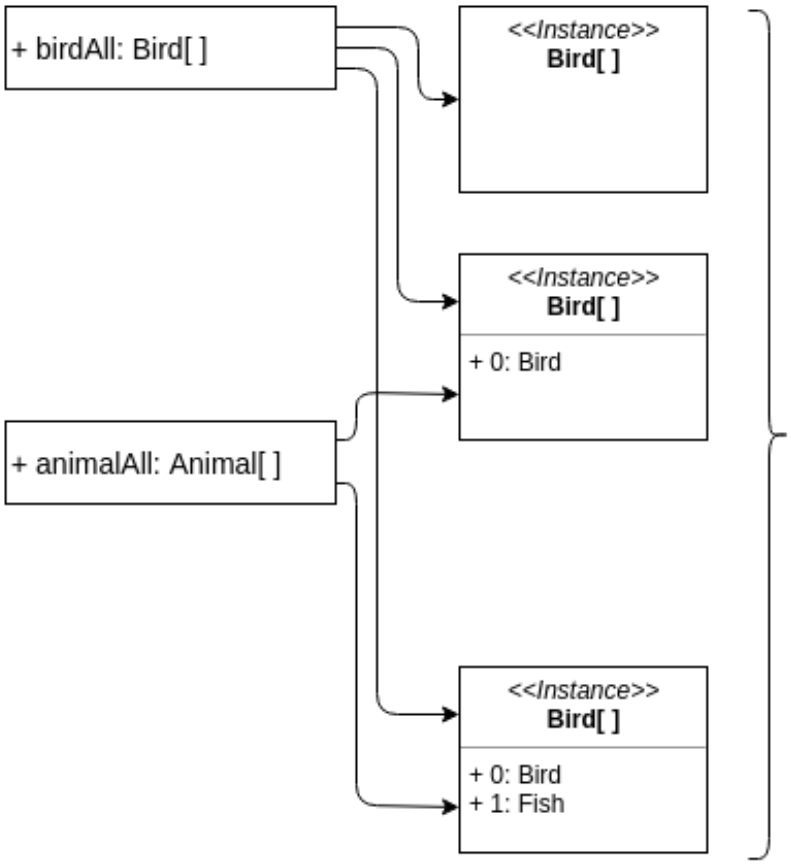
0 Создаем массив с типом Bird[ ] и присваиваем его ссылке с типом Bird[ ]

1 добавляем в массив экземпляр класса Bird

2 Присваиваем массив с типом Bird[ ] ссылке имеющей ,базовый тип Animal[ ]

3 Добавляем в массив с базовым типом Animal [ ] экземпляр класса Fish

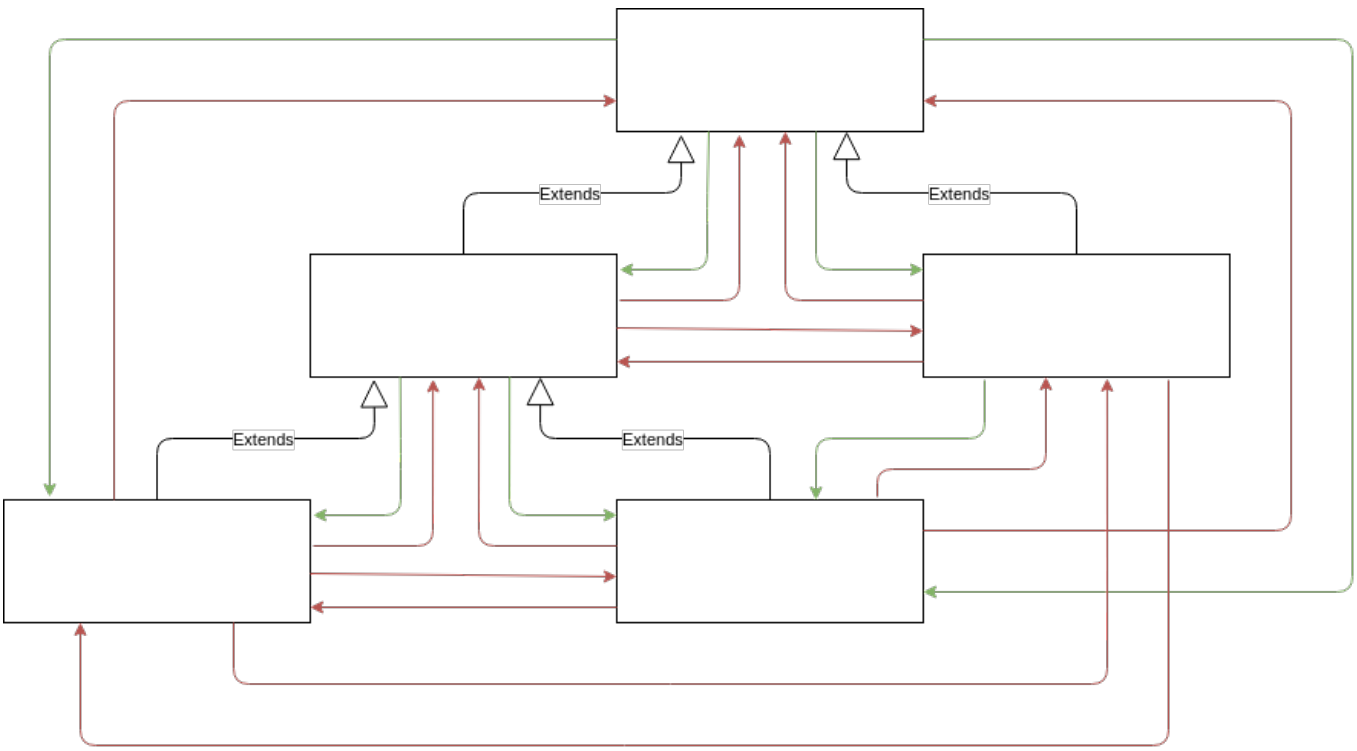
Теперь в массиве с типом Bird[ ] присутствует элемент принадлежащий к типу Fish



Ковариантность рекомендуется применять в местах, допускающих чтение.

[10.3] Контравариантность

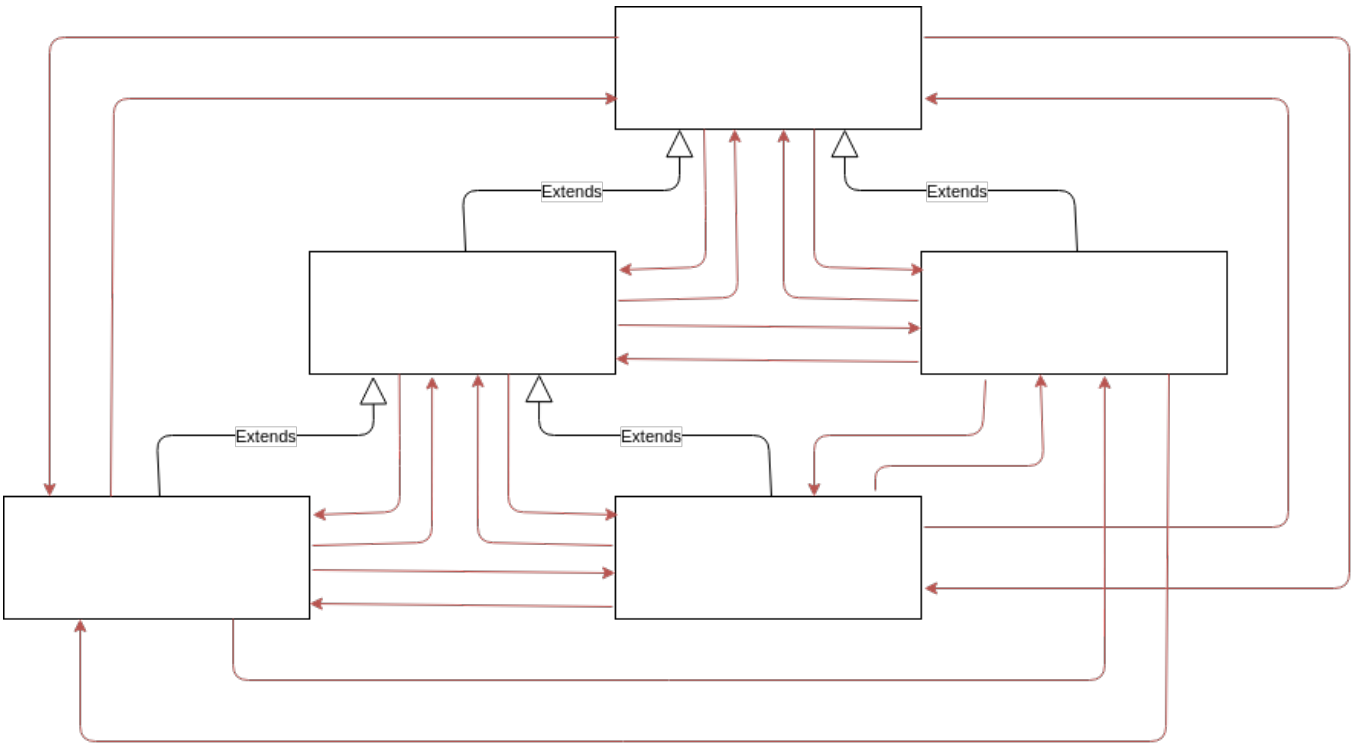
Контравариантность — это противоположный ковариантности механизм позволяющий использовать менее конкретный тип там, где изначально предполагалось использовать более конкретный тип. Другими словами, совместимыми считаются типы имеющие отношения `A < B` и `A = B`.



Контравариантность не рекомендуется в местах допускающих чтение, и наоборот, рекомендуется применять в местах допускающих запись.

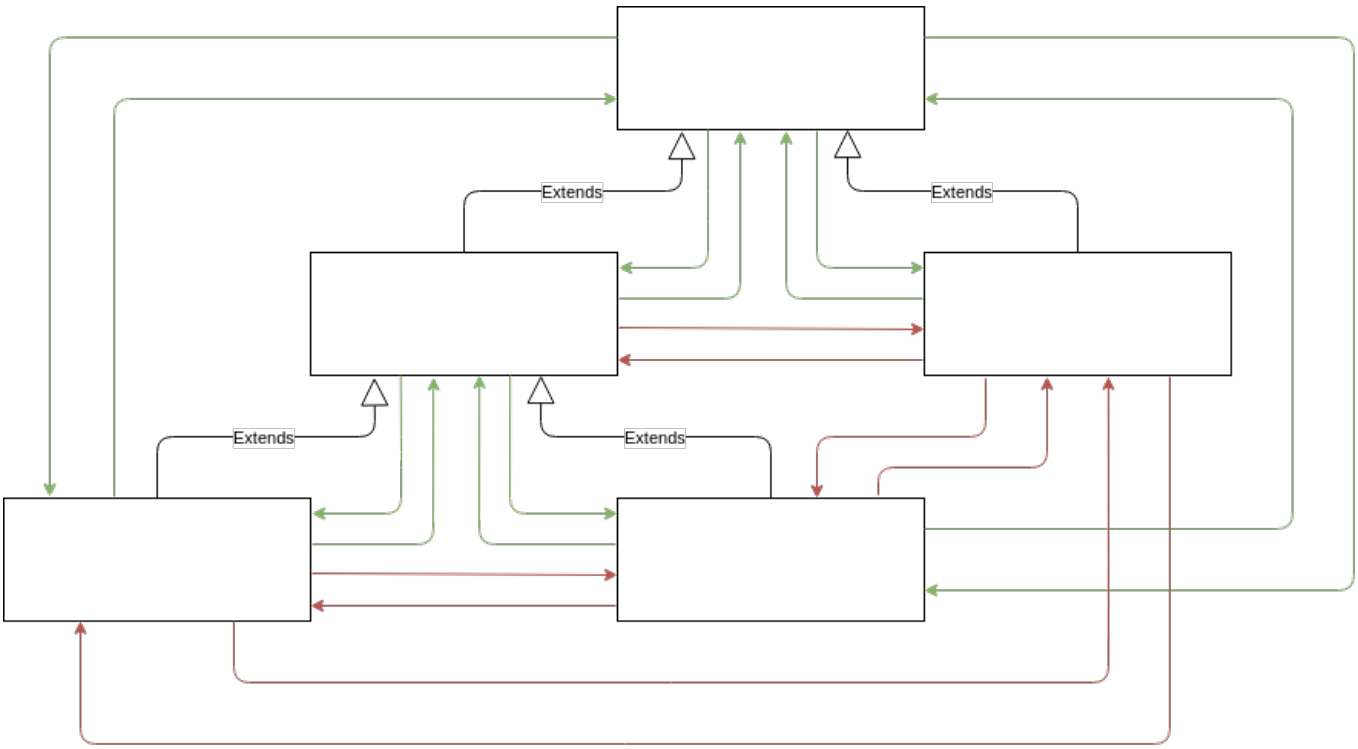
[10.4] Инвариантность

Инвариантность — это механизм позволяющий использовать только заданный тип. Совместимыми считаются только идентичные типы `A = A`.



[10.5] Бивариантность

Бивариантность — это механизм, который является представлением всех перечисленных ранее видов вариантности. В его случае совместимыми считаются любые из перечисленных ранее варианты типы `A > B`, `A < B` и `A = B`.



Бивариантность является самым не типобезопасным видом вариантности.

Глава 11

Аннотация Типов

Чтобы избавится от страха возникающего от слова *типизация*, необходимо в самом начале увидеть все преобразования которые проделал *TypeScript* над своим фундаментом коим для него является, никого не оставляющий равнодушным *JavaScript*.

[11.0] Аннотация Типов - общее

Как уже было сказано ранее, *TypeScript* — это типизированная надстройка над *JavaScript*. Другими словами *TypeScript* не добавляет никаких новых языковых конструкций (за исключением `Enum`, которая будет рассмотрена чуть позже), а лишь расширяет синтаксис *JavaScript* за счет добавления в него типов. По этой причине в этой книге не будут затрагиваться темы относящиеся к *JavaScript*, так как она рассчитана на тех, кто уже знаком с его основами. Именно поэтому погружение в типизированный мир *TypeScript* необходимо начать с рассмотрения того как типизация преобразила *JavaScript* конструкции.

[11.1] Аннотация типа

В *TypeScript* аннотация типа или указание типа осуществляется с помощью оператора двоеточия `:`, после которого следует идентификатор типа. *TypeScript* является статически типизированным языком, поэтому после того как идентификатор будет связан с типом, изменить тип будет невозможно.

[11.2] Синтаксические конструкции var, let, const

При объявлении синтаксических конструкций объявляемых с помощью операторов `var`, `let` и `const`, тип данных указывается сразу после идентификатора.

ts

```
var identifier: Type = value;
let identifier: Type = value;
const IDENTIFIER: Type = value;
```

[11.3] Функции (function)

При объявлении функции тип возвращаемого ею значения указывается между её параметрами и телом. При наличии параметров, тип данных указывается и для них.

ts

```
function identifier(param1: Type, param2: Type): ReturnedType {

}
```

Не будет лишним напомнить, что в отличие от *JavaScript*, в *TypeScript* в сигнатуру функции помимо её имени и параметров также входит и возвращаемое значение.

Помимо этого, в *TypeScript* можно объявлять параметризированные функции. Функции, имеющие параметры типа, называются обобщенными (подробнее о них речь пойдет в главе "Типы - Обобщения (Generics)"). Параметры типа заключаются в угловые скобки `<>` и располагаются перед круглыми скобками `()`, в которые заключены параметры функции.

ts
```ts
function identifier <T, U>(): ReturnedType {

}
```

Кроме того, *TypeScript* расширяет границы типизирования функций и методов с помощью незнакомого *JavaScript* разработчикам механизма *перегрузки функций*. С помощью перегрузки функций можно аннотировать функции с одинаковыми идентификаторами, но с различными сигнатурами.

Для этого перед определением функции, метода или функции-конструктора перечисляются совместимые объявления одних только сигнатур. Более подробно эта тема будет освещена позднее.

ts
```ts
function identifier(p1: T1, p2: T2): T3;
function identifier(p1: T4, p2: T5): T6;
function identifier(p1: T, p2: T): T {
    return 'value';
}

const a: T1 = 'value';
const b: T2 = 'value';
const c: T4 = 'value';
const d: T5 = 'value';

identifier(a, b); // валидно
identifier(c, d); // валидно

class Identifier {
    constructor(p1: T1, p2: T2);
    constructor(p1: T4, p2: T5);
    constructor(p1: T, p2: T) {

    }
```

```
    identifier(p1: T1, p2: T2): T3;
    identifier(p1: T4, p2: T5): T6;
    identifier(p1: T, p2: T): T {
        return 'value';
    }
}
```

[11.4] Стрелочные Функции (arrow function)

К стрелочным функциям применимы те же правила указания типов данных, что и для обычных функций, за исключением того, что возвращаемый ими тип указывается между параметрами и стрелкой.

ts

```
<T, U>(param: Type, param2: Type): Type => value;
```

[11.5] Классы (class)

Прежде чем продолжить рассмотрение изменений которые привнёс *TypeScript* в нетипизированный мир *JavaScript*, хотелось бы предупредить о том, что относительно классов будет использоваться терминология заимствованная из таких языков, как *Java* или *C#*, так как она способствует большей ясности (тем более, что в спецификации *TypeScript* встречается аналогичная терминология). Так, *переменные экземпляра* и *переменные класса* (статические переменные) в этой книге обозначаются как *поля* (*field*). *Аксессоры* (*get*_set_) обозначаются как *свойства* (*property*). А, кроме того, поля, свойства, методы, *вычисляемые свойства* (*computed property*) и *индексируемые сигнатуры* (*index signature*) обозначаются как *члены* класса (*member*).

При объявлении поля класса, как и в случае с переменными, тип данных указывается сразу после идентификатора (имени класса). Для методов класса действуют те же правила указания типов, что и для обычных функций.

Для свойств, в частности для `get`, указывается тип данных возвращаемого значения. Для `set` указывается лишь тип единственного параметра, а возвращаемый им тип и вовсе запрещается указывать явно.

Кроме того, классы в *TypeScript* также могут быть обобщенными. В случае объявления обобщенного класса, параметры типа, заключенные в треугольные скобки, указываются сразу после идентификатора класса.

ts

```ts
class Identifier<T> {
    static staticField: Type = value; // член класса

    static get staticProperty(): Type { // член класса
        return value;
    }

    static set staticProperty(value: Type) { // член класса

    }

    static staticMethod <T, U>(param0: Type, param1: Type): Type { //
член класса

    }

    static { // статический блок

    }

    [indexSignature: Type]: Type; // член класса

    [computedProp]: Type = value; // член класса

    field: Type = value; // член класса

    get property(): Type { // член класса
        return value;
    }

    set property(value: Type) { // член класса

    }

    constructor(param0: Type, param1: Type) {

    }

    method <T, U>(param0: Type, param1: Type): Type { // член класса

    }
}
```

[11.6] Сравнение Синтаксиса TypeScript и JavaScript

Перед тем, как подвести итоги этой главы, не будет лишним собрать все рассмотренные *TypeScript* конструкции и наглядно сравнить их со своими нетипизированными *JavaScript* аналогами.

ts

```ts
// .ts
var identifier: Type = value;
let identifier: Type = value;
const IDENTIFIER: Type = value;
//  .js
var identifier = value;
let identifier = value;
const IDENTIFIER = value;

// .ts
function identifier(param1: Type, param2: Type): ReturnedType {

}

// .js
function identifier(param1, param2) {

}

// .ts
class Identifier<T> {
    static staticField: Type = value;

    static get staticProperty(): Type {
        return value;
    }

    static set staticProperty(value: Type) {

    }

    static staticMethod <T, U>(param0: Type, param1: Type): Type {

    }
```

```
    [indexSignature: Type]: Type;

    [computedProp]: Type = value;

    field: Type = value;

    get property(): Type {
        return value;
    }

    set property(value: Type) {

    }

    constructor(param0: Type, param1: Type) {

    }

    method <T, U>(param0: Type, param1: Type): Type {

    }
}

// .js
class Identifier {
    static staticField = value;

    static get staticProperty() {
        return value;
    }

    static set staticProperty(value) {

    }

    static staticMethod (param, param) {

    }

    [computedProp] = value;

    field = value;

    get property() {
        return value;
    }

    set property(value) {

    }

    constructor(param0, param1) {
```

```
    }

    method (param0, param1) {

    }
}
```

[11.7] Итог

- Аннотация типа устанавливается оператором двоеточия `:`, после которого следует указание типа данных.
- При объявлении переменных тип данных указывается сразу после идентификатора.
- У функций и методов класса возвращаемый тип данных указывается между параметрами и телом.
- У стрелочных функций возвращаемый тип данных указывается между параметрами и стрелкой.
- У функций, стрелочных функций и методов класса, параметрам также указывается тип данных.
- При необходимости функциям, стрелочным функциям и методам класса можно указать параметры типа, которые заключаются в угловые скобки и указываются перед круглыми скобками, в которых размещаются параметры функции.
- В *TypeScript* аннотирование типов у функций, методов и конструкторов расширено при помощи перегрузки функций.
- Для полей класса тип данных указывается сразу после идентификатора-имени.
- Для геттеров (getters) указывается возвращаемый тип данных.
- Для сеттеров (setters) указывается тип единственного параметра и вовсе не указывается возвращаемый тип.

Глава 12

Базовый Тип Any

То, что *TypeScript* является типизированной надстройкой над *JavaScript*, от которой после компиляции не остаётся и следа, означает, что первый перенял от второго всю его идеологию. Одним из таких моментов является разделение типов данных на типы значения (примитивные) и ссылочные типы. *TypeScript* определяет неизвестный в мире *JavaScript* тип `any` являющийся базовым для всех остальных типов. Именно по этой причине с него и начнется знакомство с системой типов *TypeScript*.

[12.0] Any (any) произвольный тип

Тип `any` указывается при помощи ключевого слова `any` . Все типы в *TypeScript* являются его подтипами. Это означает, что он совместим в обе стороны с любым другим типом и с точки зрения системы типов является *высшим типом* (*top type*).

ts

```ts
let apple: any = 0;
apple = "";
apple = true;
```

Поскольку значение принадлежащие к типу `any` совместимо с любыми другими значениями, это может привести к непредсказуемым последствиям. Поэтому указывать данный тип в аннотации настоятельно рекомендуется только в самых крайних случаях. К примеру при применении техники `TDD` , которая подразумевает написание тестов раньше самого кода, тип некоторых конструкций может быть неизвестен. В таком и любых

других подобных случаях `any` допустимо указать в аннотации типа. В остальных случаях настоятельно рекомендуется рассматривать варианты с более конкретными типами.

Поскольку тип `any` позволяет работать со значением динамически, это не вызывает ошибок при обращении к неописанным в типе членам, что сводит пользу от типизации к нулю.

Примером этого может служить сервис, который работает с сервером посредством *api*. Полученные и сериализованные данные могут храниться как тип `any` прежде чем они будут и преобразованы к конкретному типу.

ts

```ts
let data: any = JSON.parse('{"id": "abc"}');
let id = data.id; // ок
```

Если при объявлении переменных и полей не было присвоено значение, компилятором будет выведен тип данных `any` .

ts

```ts
var apple; // apple: any
let lemon; // lemon: any

class Fruit {
    name; // name: any
}
```

То же правило касается и параметров функций.

ts

```ts
function weight(fruit) { // fruit: any

}
```

Кроме того, если функция возвращает значение принадлежащее к типу, который компилятор не в состоянии вывести, то возвращаемый этой функцией тип данных будет также будет выведен как тип `any` .

ts

```ts
function sum(a, b) { // function sum(a: any, b: any): any
    return a + b;
}
```

Тип `any` является уникальным для *TypeScript*, в *JavaScript* подобного типа не существует.

Глава 13

Примитивные типы Number, String, Boolean, Symbol, BigInt

Преобразования *TypeScript* не затронуло ни один из перечисленных в заголовке тип. Но несмотря на это их упоминание необходимо, поскольку их отсутствие сделало бы картину менее точной.

[13.0] Важно

Помимо того, что система типов *TypeScript* включает в себя все существующие в *JavaScript* типы данных, некоторые из них подверглись более очевидному уточнению.

Типы данных, чьи идентификаторы начинаются с прописной буквы (`Type`), представляют *объектные типы* (ссылочные типы) описывающие одноимённые типы из *JavaScript* (`Number` , `String` , `Boolean` и т.д.). Проще говоря типы знакомые по *JavaScript* и начинающиеся с большой буквы представляют конструкторы. В *TypeScript* подобные типы описаны с помощью глобальных интерфейсов (`interface`) и помимо того, что их можно расширять (`extends`) и реализовывать (`implements`) их также можно указывать в аннотации типа. Но сразу стоит сделать акцент на том, что указывать их в аннотации следует только тогда, когда подразумевается именно конструктор. Для остальных случаев существуют типы идентификаторы которых начинаются со строчной буквы (`type`). Имена типов начинающихся с маленькой буквы являются зарезервированными ключевыми словами и представляют литералы примитивных значений, то есть непосредственно числа, строки и другие значения.

Данные типы предназначены для указания в аннотациях, а не для использования в механизмах, как например наследование или расширение.

[13.1] Number (number) примитивный числовой тип

В *TypeScript*, как и в *JavaScript*, все производные от `number` являются 64-битными числами двойной точности с плавающей запятой.

Помимо того, что в *TypeScript* существует тип `Number` представляющий конструктор одноименного типа из *JavaScript*, также существует тип `number` представляющий примитивные значения числовых литералов.

ts

```ts
let v1: number; // v1: number явно
let v2 = 5.6; // v2: number неявно
```

Напомню, что числа могут записываться в двоичной, восьмеричной, десятичной, шестнадцатеричной системе счисления.

ts

```ts
let binary: number = 0b101;
let octal: number = 0o5;
let decimal: number = 5;
let hex: number = 0x5;
```

Помимо этого тип `number` неявно преобразуется в тип `Number`, но не наоборот.

ts

```ts
let n: number = 5;
let N: Number = new Number(5);

N = n; // Ok
n = N; // Error -> Type 'Number' is not assignable to type 'number'.
```

В случаях когда тип не указан явно, а в качестве значения ожидается результат вызова конструктора, то вывод типов определяет принадлежность к типу конструктора только если конструктор был вызван при помощи оператора `new`.

ts

```ts
let v0 = Number(5); // let v0: number
let v1 = new Number(5); // let v1: Number
```

В *TypeScript* поведение типа `Number` идентично поведению одноимённого типа в *JavaScript*.

[13.2] String (string) примитивный строковый тип

Примитивный тип `String` представляет собой последовательность символов в кодировке *Unicode UTF-16*. Строки могут быть заключены в одинарные или двойные кавычки, а также в обратные апострофы (инициаторы так называемых шаблонных строк).

Помимо того, что в *TypeScript* существует тип `String` описывающий одноименный конструктор из *JavaScript*, также существует тип `string` представляющий примитивные значения строковых литералов.

ts

```ts
let v1: string; // v1: string явно
let v2 = 'Bird'; // v2: string неявно
let v3: string = "Fish"; // v3: string явно
let v4: string = `Animals: ${v2}, ${v3}.`; // v4: string явно
```

Тип `string` неявно преобразуется в тип `String`, но не наоборот.

ts

```ts
let s: string = "";
let S: String = new String("");

S = s; // Ok
s = S; // Error -> Type 'String' is not assignable to type 'string'.
```

И кроме этого вывод типов выводит принадлежность к типу конструктора только если он был вызван с помощью оператора `new`.

ts

```
let v0 = String(""); // let v0: string
let v1 = new String(""); // let v1: String
```

В *TypeScript* поведение типа `String` идентично поведению одноимённого типа в *JavaScript*.

[13.3] Boolean (boolean) примитивный логический тип

Примитивный тип `Boolean` является логическим типом и представлен значениями "истина" `true` и "ложь" `false` .

Помимо того, что в *TypeScript* существует тип `Boolean` представляющий одноименный конструктор из *JavaScript*, также существует тип `boolean` представляющий примитивные значения логических литералов.

ts

```
let isV1Valid: boolean; // явно
let isV2Valid = false; // неявно
```

Тип `boolean` неявно преобразуется в тип `Boolean` , что делает его совместимым с ним, но не наоборот.

ts

```
let b: boolean = true;
let B: Boolean = new Boolean(true);

B = b; // Ok
b = B; // Error -> Type 'Boolean' is not assignable to type 'boolean'.
```

И кроме этого вывод типов выводит принадлежность к типу конструктора только если он был вызван с помощью оператора `new` .

ts

```
let v0 = Boolean(""); // let v0: boolean
let v1 = new Boolean(""); // let v1: Boolean
```

В *TypeScript* поведение типа `Boolean` идентично поведению одноимённого типа в *JavaScript*.

[13.4] Symbol (symbol) примитивный символьный тип

Примитивный тип `Symbol` предоставляет уникальные идентификаторы, которые при желании могут использоваться в качестве индексируемых членов объекта.

Помимо того, что в *TypeScript* существует тип `Symbol` описывающий одноименный конструктор из *JavaScript*, также существует тип `symbol`, представляющий примитивные значения литералов.

ts

```ts
let v1: symbol; // v1: symbol явно
let v2 = Symbol('animal'); // v2: symbol неявно
```

Тип `symbol` неявно преобразуется в тип `Symbol`, что делает его совместимым с ним, но не наоборот.

ts

```ts
let s: symbol = Symbol.for("key");
let S: Symbol = Symbol.for("key");

S = s; // Ok
s = S; // Error -> Type 'Symbol' is not assignable to type 'symbol'.
```

Поскольку конструктор `Symbol` нельзя вызвать с помощью оператора `new` вывод типов всегда будет определять принадлежность к типу `symbol`. И кроме этого вывод типов выводит принадлежность к типу конструктора только если он был вызван с помощью оператора `new`.

ts

```ts
let v0 = Symbol(""); // let v0: symbol
```

Тип `symbol` предназначен для аннотирования символьных литералов. В *TypeScript*, поведение типа `Symbol` идентично поведению одноимённого типа в *JavaScript*.

[13.5] BigInt (bigint) примитивный числовой тип

`BigInt` — примитивный числовой тип позволяющий безопасно работать с числами произвольной точности, в том числе значениями выходящими за пределы установленные типом `Number`. Примитивный тип `BigInt` указывается с помощью ключевого слова `bigint`.

ts

```ts
let bigInt: bigint = BigInt(Number.MAX_VALUE) +
BigInt(Number.MAX_VALUE);
```

Но стоит заметить, что на данный момент (конец 2018 года) из-за плохой поддержки типа `BigInt` *TypeScript* позволяет работать с ним лишь при установленной опции компилятора `--target` в значение `ESNext`.

Тип `bigint` предназначен для аннотирования числовых значений с произвольной точностью. В *TypeScript* поведение типа `BigInt` идентично поведению одноимённого типа в *JavaScript*.

Глава 14

Примитивные типы Null, Undefined, Void, Never, Unknown

Настало время рассмотреть следующую порцию типов некоторые из которых являются уникальными для *TypeScript*.

[14.0] Важно

Прежде чем приступить к знакомству с такими типами, как `Null`, `Undefined`, `Void`, `Never` и `Unknown`, стоит обговорить одну очень важную деталь. Дело в том, что все перечисленные типы можно указывать в качестве типа всем конструкциям, которые это позволяют. То есть, типом данных `null` можно аннотировать даже переменную (`let identifier: null`). Данная книга будет изобиловать подобными примерами, так как эта возможность облегчает демонстрацию совместимости типов. Но при этом стоит понимать, что проделывать подобное в реальном коде противопоказано.

[14.1] Null (null) примитивный null тип

Примитивный тип `Null` служит обозначением *“ничего”*.

Тип `Null` указывается с помощью ключевого слова `null` (не путать с единственным литеральным значением `null` типа `Null`, которое присваивается в качестве значения).

ts

```ts
let identifier: null = null; // null, указанный после оператора двоеточия, это имеющийся только в TypeScript псевдоним (alias) для глобального типа Null. В, то время как null, указанный после оператора присваивания, это единственное значение типа Null.
```

Тип `Null` является подтипом всех типов, за исключением типа `Undefined`, поэтому его единственное значение `null` совместимо со всеми остальными типами данных.

ts

```ts
class TypeSystem {
    static any: any = null; // Ok
    static number: number = null; // Ok
    static string: string = null; // Ok
    static boolean: boolean = null; // Ok
    static null: null = null; // Ok
}
```

В, то время как тип `null` совместим со всеми типами, помимо него самого, с ним самим совместим лишь тип `undefined` и `any`.

ts

```ts
TypeSystem.null = TypeSystem.any; // Ok
TypeSystem.null = TypeSystem.number; // Error
TypeSystem.null = TypeSystem.string; // Error
TypeSystem.null = TypeSystem.boolean; // Error
TypeSystem.null = TypeSystem.null; // Ok
```

Тогда, когда тип данных указывается не явно, а в качестве значения используется значение `null`, вывод типов определяет принадлежность к типу `any`.

ts

```ts
let identifier = null; // identifier: any
```

Создатели *TypeScript* во избежание ошибок возникающих при операциях в которых вместо ожидаемого значения возможно значение `null`, рекомендуют вести разработку с активным флагом `--strictNullChecks`. При активном флаге `--strictNullChecks` тип `null` является подтипом только одного типа `any`. Это в свою очередь означает, что значение `null` может быть совместимо только с типами `any` и `null`.

ts

```ts
class TypeSystem {
    static any: any = null; // Ok
    static number: number = null; // Error
    static string: string = null; // Error
    static boolean: boolean = null; // Error
    static null: null = null; // Ok
}

TypeSystem.null = TypeSystem.any; // Ok
TypeSystem.null = TypeSystem.number; // Error
TypeSystem.null = TypeSystem.string; // Error
TypeSystem.null = TypeSystem.boolean; // Error
TypeSystem.null = TypeSystem.undefined; // Ok
```

При активном флаге `--strictNullChecks`, при условии, что в качестве значения выступает значение `null`, вывод типов определяет принадлежность к типу `null`.

ts

```ts
let identifier = null; // identifier: null
```

Тип `null` идентичен по своей работе с одноимённым типом из *JavaScript*.

[14.2] Undefined (undefined) примитивный неопределенный тип

Примитивный тип `undefined` указывает на то, что значение не определено. Тип данных `undefined` указывается с помощью ключевого слова `undefined` (не путать со

свойством глобального объекта `undefined`, которое представляет единственное значение типа `Undefined`).

ts

```ts
let identifier: undefined = undefined; // undefined, указанный после оператора двоеточия, это имеющийся только в TypeScript псевдоним (alias) для глобального типа Undefined. В, то время как undefined, указанный после оператора присваивания, это единственное значение типа Undefined.
```

Во время выполнения объявленные, но не инициализированные переменные, поля и свойства класса, а также параметры имеют значение `undefined`. Также значение `undefined` является результатом вызова методов или функций, которые не возвращают значения.

Тип `undefined` является подтипом всех типов, что делает его совместимым со всеми остальными типами.

ts

```ts
class TypeSystem {
    static any: any = undefined; // Ok
    static number: number = undefined; // Ok
    static string: string = undefined; // Ok
    static boolean: boolean = undefined; // Ok
    static null: null = undefined; // Ok
    static undefined: undefined = undefined; // Ok
}
```

Может возникнуть вопрос, почему тип `null`, который не имеет непосредственного отношения к типу `undefined`, совместим с ним? На данный момент этот вопрос так и остается неразгаданным.

В, то время как тип данных `undefined` совместим со всеми типами, помимо него самого, с ним совместимы лишь `null` и `any`.

ts

```ts
TypeSystem.undefined = TypeSystem.any; // Ok
TypeSystem.undefined = TypeSystem.number; // Error
TypeSystem.undefined = TypeSystem.string; // Error
TypeSystem.undefined = TypeSystem.boolean; // Error
TypeSystem.undefined = TypeSystem.null; // Ok
```

Тогда, когда тип данных `undefined` указывается не явно, компилятор устанавливает тип `any`.

ts

```
let identifier = undefined; // identifier: any
```

При активном флаге `--strictNullChecks`, тип `undefined` является подтипом только одного типа `any`. Поэтому его и ему в качестве значения, помимо самого себя, можно присвоить только тип `any`.

ts

```ts
class TypeSystem {
    static any: any = undefined; // Ok
    static number: number = undefined; // Error
    static string: string = undefined; // Error
    static boolean: boolean = undefined; // Error
    static null: null = undefined; // Error
    static undefined: undefined = undefined; // Ok
}

TypeSystem.undefined = TypeSystem.any; // Ok
TypeSystem.undefined = TypeSystem.number; // Error
TypeSystem.undefined = TypeSystem.string; // Error
TypeSystem.undefined = TypeSystem.boolean; // Error
TypeSystem.undefined = TypeSystem.null; // Error
```

При активном флаге `--strictNullChecks`, при условии, что в качестве значения выступает значение `undefined`, вывод типов определяет принадлежность к типу `undefined`.

ts

```ts
let identifier = undefined; // identifier: undefined
```

Тип `undefined` идентичен по своей работе с одноимённым типом из *JavaScript*.

[14.3] Void (void) отсутствие конкретного типа

Тип данных `Void` можно назвать полной противоположностью типа `any`, так как этот тип означает отсутствие конкретного типа. Основное предназначение типа `Void` — явно указывать на то, что у функции или метода отсутствует возвращаемое значение.

Тип данных `Void` указывается с помощью ключевого слова `void` (не путать с одноимённым оператором из *JavaScript*) и, в отличие от таких типов, как `null` и `undefined`, не имеет никаких значений.

Тип `void` является подтипом `any` и супертипом для `null` и `undefined`.

ts

```ts
function action(): void {

}

class TypeSystem {
    static any: any = action(); // Ok
    static number: number = action(); // Error
    static string: string = action(); // Error
    static boolean: boolean = action(); // Error
    static null: null = action(); // Error
    static undefined: undefined = action(); // Error
    static void: void = action(); // Ok
}

TypeSystem.void = TypeSystem.any; // Ok
TypeSystem.void = TypeSystem.number; // Error
TypeSystem.void = TypeSystem.string; // Error
TypeSystem.void = TypeSystem.boolean; // Error
TypeSystem.void = TypeSystem.null; // Ok
TypeSystem.void = TypeSystem.undefined; // Ok
TypeSystem.void = TypeSystem.void; // Ok
```

Однако с активным флагом `--strictNullChecks`, тип данных `void` совместим лишь с `any` и `undefined`.

ts

```ts
function action(): void {

}

class TypeSystem {
    static any: any = action(); // Ok
    static number: number = action(); // Error
    static string: string = action(); // Error
    static boolean: boolean = action(); // Error
    static null: null = action(); // Error
    static undefined: undefined = action(); // Error
    static void: void = action(); // Ok
}

TypeSystem.void = TypeSystem.any; // Ok
TypeSystem.void = TypeSystem.number; // Error
TypeSystem.void = TypeSystem.string; // Error
```

```
TypeSystem.void = TypeSystem.boolean; // Error
TypeSystem.void = TypeSystem.null; // Error
TypeSystem.void = TypeSystem.undefined; // Ok
TypeSystem.void = TypeSystem.void; // Ok
```

Кому-то может показаться, что примеры чересчур излишни, или, что примеры, в которых результат вызова функции не имеющей возвращаемого значения присваивается полям с различными типами, не имеет никакого отношения к реальности. Да, это так. Но целью данных примеров является научить думать как компилятор *TypeScript*.

Когда функции в качестве возвращаемого типа указан тип `void`, может показаться, что возвращая различные значения с помощью оператора `return`, компилятор выбрасывает ошибки из-за понимания, что функция помечена как ничего не возвращающая. Но это не так. Ошибка возникает по причине несовместимости типов.

ts

```ts
function a(): void {
    let result: number = 5;

    return result; // Error
}

function b(): void {
    let result: string = '5';

    return result; // Error
}

function c(): void {
    let result: any = 5;

    return result; // Ok
}
```

Нельзя не упомянуть, что для функций и методов, которые ничего не возвращают и у которых отсутствует аннотация типа возвращаемого значения, вывод типов определяет принадлежность к типу `void`.

ts

```ts
function action() { // function action(): void

}
```

В отличие от `null` и `undefined`, тип `void` не имеет ни одного значения, которое могло бы явно продемонстрировать присвоение. Однако компилятор понимает, что имеет дело с типом `void` при вызове функции или метода, которые не возвращают значение. Этот становится ещё нагляднее, когда вывод типов устанавливает тип полученный при вызове функции или метода которые ничего не возвращают.

ts

```ts
function action(): void {

}

let identifier = action(); // identifier: void
```

Тип `void` является уникальным для *TypeScript*. В *JavaScript* подобного типа не существует.

[14.4] Never (never) примитивный тип

Примитивный типа данных `Never` служит для указания того, что какие-либо операции никогда не будут выполнены.

`Never` обозначается ключевым словом `never` и так же как и `void` не имеет явных значений.

Тип данных `never` является подтипом всех типов, что делает его совместим со всеми остальными типами.

ts

```ts
function action(): never {
    throw new Error();
};

class TypeSystem {
    static any: any = action(); // Ok
    static number: number = action(); // Ok
    static string: string = action(); // Ok
    static boolean: boolean = action(); // Ok
    static null: null = action(); // Ok
    static undefined: undefined = action(); // Ok
    static void: void = action(); // Ok
    static never: never = action(); // Ok
}

TypeSystem.never = TypeSystem.any; // Error
TypeSystem.never = TypeSystem.number; // Error
TypeSystem.never = TypeSystem.string; // Error
TypeSystem.never = TypeSystem.boolean; // Error
TypeSystem.never = TypeSystem.null; // Error
```

```
TypeSystem.never = TypeSystem.undefined; // Error
TypeSystem.never = TypeSystem.void; // Error
TypeSystem.never = TypeSystem.never; // Ok
```

Так как типу `never` нельзя присвоить значение отличное от самого типа `never`, единственным местом, в котором его может использовать разработчик является аннотация возвращаемого из функции или метода значения, с одной оговоркой. Тип `never` можно указать только той функции, из которой программа действительно никогда не сможет выйти.

Такой сценарий может выражаться в виде функции вызов которой приведет к однозначному исключению или тело функции будет включать бесконечный цикл.

ts

```ts
function error(message: string): never {
    throw new Error(message);
}

function loop(): never {
    while(true) {

    }
}
```

Вывод типов определит принадлежность возвращаемого функцией значения к типу `never` только если он указан в аннотации возвращаемого типа явно.

ts

```ts
function error(message: string): never {
    throw new Error(message);
}

function action() { // function action(): never
    return error('All very, very bad.');
}

let identifier = error(); // let identifier: never
let identifier = action(); // let identifier: never
```

Стоит заметить, что без явного указания типа `never` вывод типов определит принадлежность возвращаемого значения к типу `void`.

ts

```ts
function error(message: string) { // function error(): void
    throw new Error(message);
}
```

```
function loop() { // function loop(): void
    while(true) {

    }
}
```

Тип `never` является уникальным для *TypeScript*. В *JavaScript* подобного типа не существует.

[14.5] Unknown (unknown)

Тип `Unknown` является типобезопасным аналогом типа `any` и представлен в виде литерала `unknown` . Все типы совместимы с типом `unknown` , в, то время как сам тип `unknown` совместим только с самим собой и типом `any` .

ts

```
class TypeSystem {
    static unknown: unknown;

    static any: any = TypeSystem.unknown; // Ok
    static number: number = TypeSystem.unknown; // Error
    static string: string = TypeSystem.unknown; // Error
    static boolean: boolean = TypeSystem.unknown; // Error
    static null: null = TypeSystem.unknown; // Error
    static undefined: undefined = TypeSystem.unknown; // Error
    static void: void = TypeSystem.unknown; // Error
    static never: never = TypeSystem.unknown; // Error
}

TypeSystem.unknown = TypeSystem.any; // Ok
TypeSystem.unknown = TypeSystem.number; // Ok
TypeSystem.unknown = TypeSystem.string; // Ok
TypeSystem.unknown = TypeSystem.boolean; // Ok
TypeSystem.unknown = TypeSystem.null; // Ok
TypeSystem.unknown = TypeSystem.undefined; // Ok
TypeSystem.unknown = TypeSystem.void; // Ok
TypeSystem.unknown = TypeSystem.unknown; // Ok
```

Кроме того, над типом `unknown` запрещено выполнение каких-либо операций.

ts

```
let v0: any;
v0.a = 5; // Ok
v0.a = ''; // Ok
v0(); // Ok


let v1: unknown = v0; // Ok
v1.a = 5; // Error
v1.a = ''; // Error
v1(); // Error
```

Если тип `unknown` составляет тип пересечение (`intersection`), то он будет перекрыт всеми типами.

ts

```
type T0 = any & unknown; // type T0 = any
type T1 = number & unknown; // type T1 = number
type T2 = string & unknown; // type T2 = string
type T3 = boolean & unknown; // type T3 = boolean
type T4 = null & unknown; // type T4 = null
type T5 = undefined & unknown; // type T5 = undefined
type T6 = void & unknown; // type T6 = void
type T7 = never & unknown; // type T7 = never
type T8<T> = T & unknown; // type T8 = T
type T9 = unknown & unknown; // type T9 = unknown
```

Если тип `unknown` составляет тип объединение (`union`), то он перекроет все типы, за исключением типа `any` .

ts

```
type T0 = any | unknown; // type T0 = any
type T1 = number | unknown; // type T1 = unknown
type T2 = string | unknown; // type T2 = unknown
type T3 = boolean | unknown; // type T3 = unknown
type T4 = null | unknown; // type T4 = unknown
type T5 = undefined | unknown; // type T5 = unknown
type T6 = void | unknown; // type T6 = unknown
type T7 = never | unknown; // type T7 = unknown
type T8<T> = T | unknown; // type T8 = unknown
type T9 = unknown | unknown; // type T9 = unknown
```

Помимо этого, запрос ключей (`keyof`) для типа `unknown` возвращает тип `never` .

ts

```
type T0 = keyof number; // type T0 = "toString" | "toFixed" |
"toExponential" | "toPrecision" | "valueOf" | "toLocaleString"
```

```
type T1 = keyof any; // type T1 = string | number | symbol
type T2 = keyof unknown; // type T2 = never
```

Тип `unknown` позволяется использовать только в операциях равенства `===`, `==`, `!==` и `!=` и в операциях с логическими операторами `&&`, `||` и `!`.

ts

```
let v0: unknown = 5;

let v1 = 5 === v0; // Ok
let v2 = 5 !== v0; // Ok
let v3 = 5 > v0; // Error
let v4 = 5 < v0; // Error
let v5 = 5 >= v0; // Error
let v6 = 5 <= v0; // Error
let v7 = 5 - v0; // Error
let v8 = 5 * v0; // Error
let v9 = 5 / v0; // Error
let v10 = ++v0; // Error
let v11 = --v0; // Error
let v12 = v0++; // Error
let v13 = v0--; // Error

let v14 = 5 && v0; // Ok, let v14: unknown
let v15 = 5 || v0; // Ok, let v15: number
let v16 = v0 || 5; // Ok, let v16: unknown
let v17 = !v0; // Ok, let v17: boolean
```

Также стоит упомянуть, что функция у которой возвращаемый тип принадлежит к типу `unknown`, может не возвращать значение явно.

ts

```
function f0(): unknown {
    return; // Ok
}

function f1(): number {
    return; // Error
}

let v = f0(); // Ok, let v: unknown
```

При активной опции `--strictPropertyInitialization` принадлежащие к типу `unknown` поля не нуждаются в инициализации.

ts

```
class T {
    f0: unknown; // Ok
```

```
    f1: number; // Error
    f2: number = 5; // Ok
}
```

Если в определении типа данных участвует сопоставленный тип (`Mapped Type`) которому в качестве аргумента типа передается тип `unknown` , то такой сопоставленный тип будет выведен как объектный тип `{}` . Поскольку сопоставленные типы (`Mapped Types`), псевдонимы типов (`types`), а также обобщения (`Generics<>`) будут рассмотрены позднее, то стоит просто помнить об этом факте и повторно прочесть написанное при необходимости.

ts

```
type MappedType<T> = {
    [K in keyof T]: T;
}

type T0 =  MappedType<number>; // type T0 = number
type T1 =  MappedType<any>; // type T1 = { [x: string]: any; }
type T2 =  MappedType<unknown>; // type T2 = {}
```

Глава 15

Примитивный Тип Enum

При создании приложений тяжело обойтись без большого количества специальных конфигурационных значений. Подобные значения разработчики выносят в отдельные классы со статическими свойствами или модули с константами, избавляя таким образом свой код от *магических значений*.

TypeScript привносит новую синтаксическую конструкцию называемую `Enum` (перечисление). `enum` представляет собой набор логически связанных констант, в качестве значений которых могут выступать как числа, так и строки.

[15.0] Enum (enum) примитивный перечисляемый тип

`Enum` — это конструкция, состоящая из набора именованных констант, именуемая списком перечисления и определяемая такими примитивными типами, как `number` и `string`. `Enum` объявляется с помощью ключевого слова `enum`.

[15.1] Перечисления с числовым значением

Идентификаторы-имена для перечислений `enum` принято задавать во множественном числе. В случае, когда идентификаторам констант значение не устанавливается явно, они ассоциируются с числовым значениями, в порядке возрастания, начиная с нуля.

ts

```ts
enum Fruits {
    Apple, // 0
    Pear, // 1
    Banana // 2
}
```

Также можно установить любое значение вручную.

ts

```ts
enum Citrus {
    Lemon = 2, // 2
    Orange = 4, // 4
    Lime = 6 // 6
}
```

Если указать значение частично, то компилятор будет стараться соблюдать последовательность.

ts

```ts
enum Berries {
    Strawberry = 1,
    Raspberry, // 2

    Blueberry = 4,
    Cowberry // 5
}
```

Компилятор рассчитывает значение автоматически только на основе значения предыдущего члена перечисления. То есть, если первой и третей константе было установленно значение `10` и `20`.

ts

```
enum Keys {
  A = 10,
  B, // 11
  C = 20,
  D // 21
}
```

Поскольку `enum` позволяет разработчику задавать одинаковые значения своим константам, при частично устанавливаемых значениях нужно быть предельно внимательным, что бы не допустить ещё и повторений со стороны самого `enum` .

ts

```
enum Keys {
  A = 10,
  B, // 11
  C = 10,
  D // 11
}
```

Вдобавок ко всему `enum` позволяет задавать *псевдонимы* (*alias*). Псевдонимам устанавливается значение константы, на которую они ссылаются.

ts

```
enum Languages {
    Apple, // en, value = 0
    Apfel = Apple, // de, value = 0
    LaPomme = Apple // fr, value = 0
}
```

При обращении к константе перечисления через точечную нотацию, будет возвращено *значение*. А при обращении к перечислению с помощью скобочной нотации и указания значения в качестве ключа, будет возвращено *строковое представление идентификатора константы*.

ts

```
let value: number = Fruits.Apple; // 0
let identificator: string = Fruits[value]; // “Apple”
```

Поскольку `enum` представляет реальные значения без которых программа будет неработоспособна, он обязан оставаться в коде после компиляции. Поэтому, что бы быстрее понять `enum` , нужно посмотреть на него в скомпилированном конечном виде. Но прежде создадим его самостоятельно.

1 шаг. Тем, кто ранее работал с `enum` уже известно, что он позволяет получать строковое представление константы, а также значение ассоциированное с ней. Поэтому

для его создания требуется ассоциативный массив, коими в *JavaScript* являются объекты. Назовем объект `Fruits` и передадим его в качестве аргумента в функцию `initialization`, которая будет содержать код его инициализации.

ts
```ts
let Fruits = {};

function initialization(Fruits){

}
```

2 шаг. Создадим поле с именем `Apple` и присвоим ему в качестве значения число `0`.

ts
```ts
let Fruits = {};

function initialization(Fruits) {
    Fruits["Apple"] = 0;
}
```

3 шаг. Ассоциация константа-значение произведена, осталось создать зеркальную ассоциацию значение-константа. Для этого создадим ещё одно поле у которого в качестве ключа будет выступать значение `0`, а в качестве значения — строковое представление константы, то есть имя.

ts
```ts
let Fruits = {};

function initialization(Fruits) {
    Fruits[ "Apple" ] = 0;
    Fruits[ 0 ] = "Apple";
}
```

4 шаг. Теперь сократим код, но начала вспомним, что результатом операции присваивания является значение правого операнда. Поэтому сохраним результат первого выражения в переменную `value`, а затем используем её в качестве ключа во втором выражении.

ts
```ts
let Fruits = {};

function initialization(Fruits) {
    let value = Fruits["Apple"] = 0; //, то же самое, что value = 0
    Fruits[value] = "Apple"; //, то же самое, что Fruits[0] = "Apple";
}
```

5 шаг. Продолжим сокращать и в первом выражении откажемся от переменной `value`, а во втором выражении на её место поместим первое выражение.

ts

```ts
let Fruits = {};

function initialization( Fruits ){
    Fruits[Fruits["Apple"] = 0] = "Apple";
}
```

6 шаг. Теперь проделаем, то же самое для двух других констант.

ts

```ts
let Fruits = {};

function initialization(Fruits) {
    Fruits[Fruits["Apple"] = 0] = "Apple";
    Fruits[Fruits["Lemon"] = 1] = "Lemon";
    Fruits[Fruits["Orange"] = 2] = "Orange";
}
```

7 шаг. Теперь превратим функции `initialization` в самовызывающееся функциональное выражение и лучше анонимное.

ts

```ts
let Fruits = {};

(function(Fruits) {
    Fruits[Fruits["Apple"] = 0] = "Apple";
    Fruits[Fruits["Pear"] = 1] = "Pear";
    Fruits[Fruits["Banana"] = 2] = "Banana";
})(Fruits);
```

8 шаг. И перенесем инициализацию объекта прямо на место вызова.

ts

```ts
let Fruits;
(function(Fruits) {
    Fruits[Fruits["Apple"] = 0] = "Apple";
    Fruits[Fruits["Pear"] = 1] = "Pear";
    Fruits[Fruits["Banana"] = 2] = "Banana";
})(Fruits || (Fruits = {}));
```

Перечисление готово. Осталось сравнить созданное перечисление с кодом полученным в результате компиляции.

ts

```ts
// enum сгенерированный typescript compiler
let Fruits;
(function (Fruits) {
  Fruits[Fruits["Apple"] = 0] = "Apple";
  Fruits[Fruits["Pear"] = 1] = "Pear";
  Fruits[Fruits["Banana"] = 2] = "Banana";
})(Fruits || (Fruits = {}));
```

Теперь добавим в рассматриваемое перечисление псевдоним `LaPomme` (яблоко на французском языке) для константы `Apple`.

ts

```ts
enum Fruits {
    Apple, // 0
    Pear, // 1
    Banana, // 2

    LaPomme = Apple // 0
}
```

И снова взглянем на получившийся в результате компиляции код. Можно увидеть, что псевдоним создается так же, как обычная константа, но в качестве значения ему присваивается значение идентичное константе на которую он ссылается.

ts

```ts
(function (Fruits) {
  Fruits[Fruits["Apple"] = 0] = "Apple";
  Fruits[Fruits["Lemon"] = 1] = "Lemon";
  Fruits[Fruits["Orange"] = 2] = "Orange";
  Fruits[Fruits["LaPomme"] = 0] = "LaPomme"; // псевдоним
})(Fruits || (Fruits = {}));
```

[15.2] Перечисления со строковым значением

Помимо значения принадлежащего к типу `number`, *TypeScript* позволяет указывать значения с типом `string`.

ts

```
enum FruitColors {
    Red = "#ff0000",
    Green = "#00ff00",
    Blue = "#0000ff"
}
```

Но в случае, когда константам присваиваются строки, ассоциируется только ключ со значением. Обратная ассоциация (значение-ключ) — отсутствует. Простыми словами, по идентификатору (имени константы) можно получить строковое значение, но по строковому значению получить идентификатор (имя константы) невозможно.

ts

```
var FruitColors;
(function (FruitColors) {
    FruitColors["Red"] = "#ff0000";
    FruitColors["Green"] = "#00ff00";
    FruitColors["Blue"] = "#0000ff";
})(FruitColors || (FruitColors = {}));
```

Тем не менее остается возможность создавать *псевдонимы* (*alias*).

ts

```
enum FruitColors {
    Red = "#ff0000",
    Green = "#00ff00",
    Blue = "#0000ff",

    Rouge = Red, // fr "#ff0000"
    Vert = Green, // fr "#00ff00"
    Bleu = Blue // fr "#0000ff"
}
```

И снова изучим скомпилированный код. Можно убедится, что псевдонимы создаются так же, как и константы. А значение присваиваемое псевдонимам идентично значению констант на которые они ссылаются.

ts

```
var FruitColors;
(function (FruitColors) {
    FruitColors["Red"] = "#ff0000";
    FruitColors["Green"] = "#00ff00";
    FruitColors["Blue"] = "#0000ff";
    FruitColors["Rouge"] = "#ff0000";
    FruitColors["Vert"] = "#00ff00";
    FruitColors["Bleu"] = "#0000ff";
})(FruitColors || (FruitColors = {}));
```

[15.3] Смешанное перечисление (mixed enum)

Если в одном перечислении объявлены числовые и строковые константы, то такое перечисление называется *смешанным* (*mixed enum*).

Со смешанным перечислением связаны две неочевидные особенности.

Первая из них заключается в том, что константам, которым значение не задано явно, присваивается числовое значение по правилам перечисления с числовыми константами.

ts

```ts
enum Stones {
    Peach, // 0
    Apricot = "apricot"
}
```

Вторая особенность заключается в том, что если константа, которой значение не было присвоено явно, следует после константы со строковым значением, то такой код не скомпилируется. Причина заключается в том, что как было рассказано в главе *“Перечисления с числовым значением”*, если константе значение не было установлено явно, то её значение будет рассчитано, как значение предшествующей ей константе `+1`, либо `0`, в случае её отсутствия. А так как у предшествующей константы значение принадлежит к строковому типу, то рассчитать число на его основе не представляется возможным.

ts

```ts
enum Stones {
    Peach, // 0
    Apricot = "apricot",
    Cherry, // Error
    Plum // Error
}
```

Для разрешения этой проблемы в смешанном перечислении, константе, которая была объявлена после константы со строковым значением, необходимо задавать значение явно.

ts

```ts
enum Stones {
    Peach, // 0
```

```
    Apricot = "apricot",
    Cherry = 1, // 1
    Plum // 2
}
```

[15.4] Перечисление в качестве типа данных

Может возникнуть мысль использовать перечисление в качестве типа данных переменной или параметра. Это вполне нормальное желание, но нужно быть очень осторожным: в *TypeScript* с перечислением связан один достаточно неприятный нюанс. Дело в том, что пока в перечислении есть хотя бы одна константа с числовым значением, он будет совместим с типом `number`. Простыми словами, любое число проходит проверку совместимости типов с любым перечислением.

Функцию, тип параметра которой является смешанным перечислением, благополучно получится вызвать как с константой перечисления в качестве аргумента, так и с любым числом. Вызвать эту же функцию с идентичной константе перечисления строкой уже не получится.

ts

```ts
enum Fruits {
    Apple,
    Pear,
    Banana = "banana"
}

function isFruitInStore(fruit: Fruits): boolean {
    return true;
}

isFruitInStore(Fruits.Banana); // ок
isFruitInStore(123456); // ок
isFruitInStore("banana"); // Error
```

Если перечисление содержит константы только со строковыми значениями, то совместимыми считаются только константы перечисления указанного в качестве типа.

ts

```ts
enum Berries {
    Strawberry = "strawberry",
    Raspberry = "raspberry",
    Blueberry = "blueberry"
```

```
}

function isBerryInStory(berry: Berries): boolean {
    return true;
}

isBerryInStory(Berries.Strawberry); // ок
isBerryInStory(123456); // Error
isBerryInStory("strawberry"); // Error
```

Поведение не совсем очевидное, поэтому не стоит забывать об этом при использовании перечислений в которых присутствуют константы с числовым значением в качестве типа.

[15.5] Перечисление const с числовым и строковым значением

Перечисление `enum` объявленное с помощью ключевого слова `const` после компиляции не оставляет в коде привычных конструкций. Вместо этого компилятор встраивает литералы значений в места, в которых происходит обращение к значениям перечисления. Значения констант перечисления могут быть как числовыми, так и строковыми типами данных. Так же, как и в обычных перечислениях, в перечислениях объявленных с помощью ключевого слова `const`, есть возможность создавать *псевдонимы* (*alias*) для уже объявленных констант.

Если создать два перечисления `Apple` и `Pear`, у каждого из которых будет объявлена константа `Sugar` с числовым значением, то на основе этих констант можно рассчитать количество сахара в яблочно-грушевом соке. Присвоив результат операции сложения количества сахара в промежуточную переменную, мы получим хорошо читаемое, задекларированное выражение.

ts

```
const enum Apple {
    Sugar = 10
}

const enum Pear {
    Sugar = 10
}

let calciumInApplePearJuice: number = Apple.Sugar + Pear.Sugar;
```

После компиляции от перечисления не остается и следа, так как константы будут заменены числовыми литералами. Такое поведение называется *inline встраивание*.

ts

```ts
let calciumInApplePearJuice = 10 + 10;
```

Обращение к значению через точечную нотацию требует большего времени, чем обращение к литеральному значению напрямую. Поэтому код с inline конструкциями выполняется быстрее по сравнению с кодом, в котором происходит обращение к членам объекта. Прибегать к подобному подходу рекомендуется только в тех частях кода, которые подвержены высоким нагрузкам. За счет перечисления, объявленного с ключевым словом `const`, исходный код будет легко читаемым, а конечный код — более производительным.

Тип `enum` является уникальным для *TypeScript*, в *JavaScript* подобного типа не существует.

[15.6] Когда стоит применять enum?

Может возникнуть вопрос - *"Когда использовать enum и стоит ли это делать с учетом закрепившейся привычки работы со статическими классами и константами?"*.

Ответ очевиден — безусловно стоит применять тогда, когда нужна двухсторонняя ассоциация строкового ключа с его числовым или строковым значением (проще говоря, карта *строковый ключ — числовое значение*_числовой ключ — строковое значение_).

Кроме того, `enum` лучше всего подходит для определения *дискриминантных полей* речь о которых пойдет позже.

Ну а тем, кто считает, что скомпилированная конструкция `enum` отягощает их код и при этом они пользовались ранее транскомпилятором `Babel`, то ответьте себе на вопрос: *"Почему вы это делали, если он добавляет в сотню раз больше лишнего кода?"*. Рассуждение о том, что несколько лишних строк кода испортит или опорочит программу, является пустой тратой драгоценного времени.

Поэтому если есть желание использовать `enum`, то делайте это. Мне не доводилось встречать приложения, в которых не было бы `enum`, константных классов и просто модулей с константами одновременно. И это более чем нормально.

Глава 16

Типы - Union, Intersection

Чем в *TypeScript* можно заменить наследование? Как указать сразу диапазон типов? Ответы на эти вопросы вы сможете получить прочитав до конца данную главу.

[16.0] Тип Объединение (Union Types)

Объединение (`Union`) — это мощный механизм позволяющий создавать из множества существующих типов логическое условие по которому данные могут принадлежать только к одному из указанных типов. Объединение указывается с помощью оператора прямой черты `|` , по обе стороны которой располагаются типы данных.

ts

```ts
let v1: T1 | T2 | T3;
```

Переменной, которой был указан тип объединения `A` или `B` или `C` , может быть присвоено значение, принадлежащие к одному из трех типов.

ts

```ts
class A {
  a: number;
}
class B {
  b: string;
}
```

```
class C {
  c: boolean;
}

let identifier: A | B | C; // значение может принадлежать только одному
типу (A или B или C)
```

Поскольку значение может принадлежать ко множеству порой несовместимых типов, компилятор, без вмешательства разработчика, то есть без конкретизации типа, определяет принадлежность значения к типу который наделен общими для всех типов признаками.

ts

```
class Bird {
    fly(): void {}

    toString(): string {
        return 'bird';
    }
}

class Fish {
    swim(): void {}

    toString(): string {
        return 'fish';
    }
}

class Insect {
    crawl(): void {}

    toString(): string {
        return 'insect';
    }
}

function move(animal: Bird | Fish | Insect): void {
    animal.fly(); // Error
    animal.swim(); // Error
    animal.crawl(); // Error

    animal.toString(); // ок, задекларировано во всех типах
}
```

[16.1] Тип Пересечение (Intersection Type)

Пересечение (`Intersection`) — ещё один мощный механизм *TypeScript*, который позволяет рассматривать множество типов данных как единое целое. Пересечение указывается с помощью оператора амперсанда `&` по обе стороны от которого указываются типы данных.

ts

```ts
let v1: T1 & T2 & T3;
```

Переменной, которой был указан тип пересечение `A` и `B` и `C` должно быть присвоено значение принадлежащее к типам `A` и `B` и `C` одновременно. Другими словами значение должно обладать всеми *обязательными* признаками каждому типа определяющего пересечение.

ts

```ts
class A {
  a: number;
}
class B {
  b: string;
}
class C {
  c: boolean;
}

let name: A & B & C; // значение должно принадлежать ко всем типам
одновременно
```

Глава 17

Type Queries (запросы типа), Alias (псевдонимы типа)

Как сначала определить сложное значение, а затем одной строкой описать его тип? Или как конкретизировать более общий идентификатор типа и тем самым увеличить семантическую привлекательность кода? На два эти важные вопроса и поможет ответить текущая глава.

[17.0] Запросы Типа (Type Queries)

Механизм *запроса типа* позволяют получить тип связанный со значением по его идентификатору и в дальнейшим использовать его как обычный тип. Запрос типа осуществляется оператором `typeof` после которого идет идентификатор ссылающийся на значение. Запрос типа также может располагаться в местах указания типа.

ts

```ts
let v1: T1;
let v2: typeof v1; // let v2: T1;
```

С помощью данного механизма можно получить тип любой конструкции будь, то переменная, параметр функции или метода, а также членов объекта и класса.

ts

```ts
class T {
    static staticProp: number;

    field: string;

    get prop(): boolean {
        return true;
    }

    method(): void {

    }
}

let t: T = new T();

let v0: typeof t; // let v0: T
let v1: typeof T.staticProp; // // let v1: number
let v2: typeof t.field; // let v2: string
let v3: typeof t.prop; // let v3: boolean
let v4: typeof t.method; // let v4: ()=>void

function f(param: number): void {
    let v: typeof param; // let v: number
}
```

Запрос типа может быть очень полезен сторонникам минимализма достигаемого при помощи вывода типов. К слову я один из них. Тем, кто придерживается консерватизма, возможно придется по душе идея ускорять написание тестов за счет механизма вывода типов. Ведь в тех ситуациях, когда для тестирования требуются не определенные в приложении типы данных, часто не хочется тратить время на их декларацию, но при этом хочется использовать авто дополнение. Например, при тестировании метода класса может понадобиться тип представляющий только его, но поскольку для проекта подобный тип просто бессмыслен, его придется определять в контексте самих тестов, что гораздо проще сделать при помощи механизма запроса типа. Все это вместе в *TypeScript* становится возможным благодаря выводу типов в паре с оператором запроса типа.

Представьте значение, присвоенное переменной, тип которой не указан явно. Теперь представьте, что это значение нужно передать в функцию, параметр которой также не имеет явного указания типа. В этом случае в функции будет сложно работать с параметрами, так как вывод типов определит его принадлежность к типу `any`.

ts

```ts
const STANDARD_NORMAL = { x: 0, y: 0 }; // данные, которые нужны только
для контролирования точности самих тестов. А это, в свою очередь,
означает, что декларация типов для них ещё не определена. Хотя вывод
типов в состоянии вывести тип {x: number, y: number} для этой константы.

// здесь вывод типа не в состоянии вывести тип параметров функции
function valid(standard) {
```

```
    let element = document.querySelector('#some-id');
    let { clientLeft: x, clientTop: y } = element;
    let position = { x, y };

    // поэтому о параметрах невозможно получить какую-либо информацию
    let isPositionXValid = position.x === standard. // автодополнение
отсутствует
    let isPositionYValid = position.y === standard. // автодополнение
отсутствует

    // ...
}
```

Не стоит даже рассуждать — оставить так или указать типы, которые, возможно, предварительно нужно ещё задекларировать. Вместо этого нужно прибегнуть к механизму *запроса типа*. Запрос типа позволяет одновременно решить две задачи, одна из которых связана с проблемами сопутствующими типу `any`, а другая — минимализму и расходу времени на декларирование типов.

ts

```ts
const STANDARD_NORMAL = { x: 0, y: 0 };

// получение типа для аннотирования параметров прямо из константы.
function valid(standard: typeof STANDARD_NORMAL) {
    let element = document.querySelector('#some-id');
    let { clientLeft: x, clientTop: y } = element;
    let position = { x, y };

    // расходовать время на декларацию типа так и не пришлось. Тем не
менее автодополнение работает.
    let isPositionXValid = position.x === standard.x; // выводит .x
    let isPositionYValid = position.y === standard.y; // выводит .y

    // ...
}
```

[17.1] Псевдонимы Типов (Type Aliases)

Создание *псевдонимов типа* (*types alias*) — ещё одна из множества возможностей *TypeScript* которые невозможно переоценить. Псевдоним типа объявляется при помощи ключевого слова `type`, после которого следует идентификатор (имя) псевдонима, а за ним идет оператор присваивания `=`, справа от которого находится тип, ассоциирующийся с псевдонимом.

ts

```ts
type Alias = T1;
```

Объявляться псевдоним типа может в контексте модулей, функций и методов.

ts

```ts
class Type {
    method(): void {
        type Alias = Type;
    }
}

type Alias = Type;

function func(): void {
    type Alias = Type;
}
```

Так как псевдонимы типов являются лишь псевдонимами для реальных типов, они не оставляют следа в коде после компиляции, к тому же их нельзя было расширять (`extends`) и реализовать (`implements`) в ранних версиях языка (до 2.7). Сейчас псевдоним типа можно реализовать или расширить, только если он представляет объектный тип (`object type`) или пересечение объектных типов со статически известными членами. Кроме того, псевдонимы типов нельзя использовать в таких операциях с типами времени выполнения как `typeof` и `instanceof` . Если псевдоним типа будет создан для объекта, то при попытке создать его экземпляр возникнет ошибка.

ts

```ts
class Class {
    f1: number;
    f2: string;
}

type ClassAlias = Class;

let v1: ClassAlias = new Class(); // Ok
let v2: ClassAlias = new ClassAlias(); // Error
```

Псевдонимы типов можно создавать как для типов объединений, так и для типов пересечений.

ts

```ts
type SomeType = number | string | boolean; // union type
type OtherType = number & string & boolean; // intersection type
```

Давно доказано, что идентификаторы типов, которые однозначно говорят о своем предназначении, облегчают понимание кода и его поддержку и тем самым сокращая затраты на его написание. По этой причине имена типов могут получаться очень длинными. Создание объединений или пересечений из нескольких типов с длинными именами может привести к ситуации, при которой код не поместится на одной строчке, что приведет к обратному эффекту, то есть затруднит его чтение.

ts

```ts
class BirdSignDataProvider {}
class FishSignDataProvider {}
class InsectSignDataProvider {}

function animalSignValidate(
    signProvider: BirdSignDataProvider | FishSignDataProvider |
InsectSignDataProvider
): boolean {
  return true;
}
```

При работе с типами объединения и пересечения псевдонимы типов позволяют повысить читаемость кода за счет сокрытия множества типов за одним компактным идентификатором.

ts

```ts
class BirdSignDataProvider {};
class FishSignDataProvider {};
class InsectSignDataProvider {};

type AnimalSignProvider =
    BirdSignDataProvider |
    FishSignDataProvider |
    InsectSignDataProvider;

function animalSignValidate(signProvider: AnimalSignProvider): boolean {
    return true;
}
```

Псевдонимы типов можно выносить в отдельные модули, а затем импортировать их в места назначения. Если модуль содержащий псевдонимы типов содержит только их, современные сборщики не будут включать такой модуль в сборку. Другими словами, модуль растворится точно так же, как и другие не имеющие место в *JavaScript* конструкции *TypeScript*.

ts

```ts
// aliases.ts
import BirdSignDataProvider from './BirdSignDataProvider';
import FishSignDataProvider from './FishSignDataProvider';
import InsectSignDataProvider from './InsectSignDataProvider';
```

```
export type AnimalSignProvider =
    BirdSignDataProvider |
    FishSignDataProvider |
    InsectSignDataProvider;


// index.js
import { AnimalSignProvider } from './aliases';

import BirdSignDataProvider from './BirdSignDataProvider';
import FishSignDataProvider from './FishSignDataProvider';
import InsectSignDataProvider from './InsectSignDataProvider';

function animalSignValidate(signProvider: AnimalSignProvider): boolean {
    return true;
}

animalSignValidate(new BirdSignDataProvider());
animalSignValidate(new FishSignDataProvider());
animalSignValidate(new InsectSignDataProvider());
```

Как было сказано ранее в главе *“Псевдонимы Типов (Type Aliases)”*, в тех редких случаях, когда декларированием типов, требующихся только для тестирования, можно пренебречь, механизм запроса типов помогает получить тип для указания в аннотации типа прямо из значения. Это дает все возможности типизации, за исключением читаемости кода, поскольку выражение запроса не персонализирует полученный тип. Хотя в примере, иллюстрирующим работу механизма запроса типа, константа `STANDARD_NORMAL` имеет вполне говорящий идентификатор, допускаются случаи, при которых подобного будет сложно добиться. При худшем сценарии идентификатор может иметь общий смысл.

ts

```
let data = { x: 0, y: 0 };

function valid(standard: typeof data) { // data, что это?

}
```

В таких случаях псевдоним типа может оказать неоценимую помощь. Ведь экономия времени затраченного на декларирование типов не лишит код его выразительности и семантики.

ts

```
const STANDARD_NORMAL = { x: 0, y: 0 };

type StandardNormalPoint = typeof STANDARD_NORMAL; // определение
"говорящего типа" без затраты времени на его декларирование.
```

```
function valid(standard: StandardNormalPoint) {
    // ...

    // Расходовать время на декларацию типа не пришлось, при этом
работает автодополнение и параметр функции обзавелся типом, в чьем
названии заключено его предназначение.
    let isPositionXValid = position.x === standard.x; // выводит .x
    let isPositionYValid = position.y === standard.y; // выводит .y

    // ...
}
```

Есть ещё пара особенностей псевдонимов, указание которых в данной главе было бы преждевременно, поэтому здесь о них будет лишь упомянуто. Во-первых, вынести объявления кортежа (`Tuple`), речь о котором пойдет далее в главе “Типы - Object, Array, Tuple”, можно только в описание псевдонима. Во-вторых, создать тип сопоставления как, например, `Readonly` , `Partial` , `Pick` , `Record` и им подобных, можно исключительно на основе псевдонимов типов. Перечисленные типы будут подробно рассмотрены в главе “Расширенные типы - Readonly, Partial, Required, Pick, Record”.

Глава 18

Примитивные литеральные типы Number, String, Template String, Boolean, Unique Symbol, Enum

Помимо обычных примитивных типов перешедших их *JavaScript*, в *TypeScript* существуют так называемые *литеральные типы*, которые, как можно понять из названия, представляют литералы обычных примитивных типов. Число `5`, строка `“apple”`, логическое значение `true` или константа перечисления `Fruits.Apple` может выступать в качестве самостоятельно типа. Не сложно догадаться, что в качестве значений в таком случае могут выступать только литеральные эквиваленты самих типов, а также `null` и `undefined` (при `--strictNullChecks` со значением `false`).

Литеральные типы были созданы для того, что бы на этапе компиляции выявлять ошибки, возникающие из-за несоответствия значений заранее объявленных констант, как, например, номер порта или идентификатор динамического типа. Ранее такие ошибки можно было выявить только на этапе выполнения.

[18.0] Литеральный тип Number (Numeric Literal Types)

Литеральный тип `number` должен состоять из литеральных значений, входящих в допустимый диапазон чисел от `Number.MIN_VALUE` (-9007199254740992) до `Number.`

`MAX_VALUE` (9007199254740992), и может записываться в любой системе счисления (двоичной, восьмеричной, десятичной, шестнадцатеричной).

Очень часто в программе фигурируют константные значения, ограничить которые одним типом недостаточно. Здесь на помощь и приходят литеральные типы данных. Сервер, конфигурацией которого разрешено запускаться на `80` или `42` порту, мог бы иметь метод, вызываемый с аргументами, включающими номер порта. Поскольку значение принадлежало бы к типу `number`, то единственный способ его проверки был возможен при помощи условия расположенном в блоке `if`, с последующим выбрасыванием исключения. Но подобное решение выявило бы несоответствие только на этапе выполнения. Помощь статической типизации в данном случае выражалась лишь в ограничении по типу `number`.

ts

```ts
const port80: number = 80;
const port42: number = 42;

// параметры ограничены лишь типом данных
function start(port: number): void {
    // блок if сообщит об ошибке только во время выполнения
    if (port !== port80 || port !== port42) {
        throw new Error(`port #${port} is not valid.`);
    }
}

start(81); // вызов с неправильным значением
```

Именно для таких случаев и были введены литеральные типы данных. Благодаря литеральному типу `number` стало возможно выявлять ошибки не дожидаясь выполнения программы. В данном случае значение допустимых портов можно указать в качестве типа параметров функции.

ts

```ts
const port80: number = 80;
const port42: number = 42;

function start(port: 80 | 42): void {
    // блок if сообщит об ошибке только во время выполнения
    if (port !== port80 || port !== port42) {
        throw new Error(`port #${port} is not valid.`);
    }
}

start(81); // ошибка выявлена на этапе компиляции!
```

Для повышения семантики кода литеральные типы, представляющие номера порта, можно сокрыть за псевдонимом типа.

ts

```ts
type ValidPortValue = 80 | 42;

const port80: number = 80;
const port42: number = 42;

function start(port: ValidPortValue): void {
    // блок if сообщит об ошибке только во время выполнения
    if (port !== port80 || port !== port42) {
        throw new Error(`port #${port} is not valid.`);
    }
}

start(81); // ошибка выявлена на этапе компиляции!
```

Как уже было сказано ранее, литеральный тип `number` можно указывать в любой системе счисления.

ts

```ts
type NumberLiteralType = 0b101 | 0o5 | 5 | 0x5;
```

Примитивный литеральный тип `number` является уникальным для *TypeScript*, в *JavaScript* подобного типа не существует.

[18.1] Литеральный тип String (String Literal Types)

Литеральный тип `string` может быть указан только строковыми литералами, заключенными в одинарные (`' '`) или двойные (`" "`) кавычки. Так называемые шаблонные строки, заключенные в обратные кавычки (`` ` ` ``), не могут быть использованы в качестве строкового литерального типа.

В ходе разработки, конвенциями проекта могут быть наложены ограничения на типы используемой анимации. Чтобы не допустить ошибочных идентификационных значений, их тип можно ограничить литеральным строковым типом.

ts

```ts
function animate(name: "ease-in" | "ease-out"): void {

}
```

```
animate('ease-in'); // Ok
animate('ease-in-out'); // Error
```

Примитивный литеральный тип `string` является уникальным для *TypeScript*, в *JavaScript* подобного типа не существует.

[18.2] Шаблонный литеральный тип String (Template String Literal Types)

Шаблонный литеральный строковый тип — это тип, позволяющий на основе литеральных строковых типах динамически определять новый литеральный строковый тип. Простыми словами, это известный по *JavaScript* механизм создания шаблонных строк только для типов.

ts

```ts
type Type = "Type";
type Script = "Script";

/**
 * type Message = "I ❤️ TypeScript"
 */
type Message = `I ❤️ ${Type}${Script}`;
```

Но вся мощь данного типа раскрывается в момент определение нового типа на основе объединения (`union`). В подобных случаях новый тип будет также представлять объединение, элементы которого представляют все возможные варианты, полученные на основе исходного объединения.

ts

```ts
type Sides = "top" | "right" | "bottom" | "left";

/**
 * type PaddingSides = "padding-top" | "padding-right" | "padding-
bottom" | "padding-left"
 */
type PaddingSides = `padding-${Sides}`;
```

Аналогичное поведение будет справедливо и для нескольких типов объединения.

ts

```ts
type AxisX = "top" | "bottom";
type AxisY = "left" | "right";


/**
 * type Sides = "top-left" | "top-right" | "bottom-left" | "bottom-
right"
 */
type Sides = `${AxisX}-${AxisY}`;

/**
 * type BorderRadius = "border-top-left-radius" | "border-top-right-
radius" | "border-bottom-left-radius" | "border-bottom-right-radius"
 */
type BorderRadius = `border-${Sides}-radius`;
```

Поскольку с высокой долей вероятности в подобных операциях потребуется трансформация регистра строк, создателями данного механизма также были добавлены новые утилитарные алиасы типов - `Uppercase`, `Lowercase`, `Capitalize` и `Uncapitalize`.

ts

```ts
type A = `${Uppercase<"AbCd">}`; // type A = "ABCD"
type B = `${Lowercase<"AbCd">}`; // type B = "abcd"
type C = `${Capitalize<"abcd">}`; // type C = "Abcd"
type D = `${Uncapitalize<"Abcd">}`; // type D = "abcd"
```

Помимо этого, компилятор _TypeScript _ умеет понимать, что строка объявленная с помощью косых кавычек и сформированная при помощи значения ассоциированного с переменной, совместима с шаблонным строковым литеральным типом.

ts

```ts
/**
 * [*] Ok!
 */
function f(param: string): `Hello ${string}` {
  return `Hello ${param}`; // [*]
}
```

[18.3] Литеральный Тип Boolean (Boolean Literal Types)

Литеральный тип `boolean` ограничен всего двумя литеральными значениями `true` и `false`.

Так как литеральный тип `boolean` состоит всего из двух литеральных значений `true` и `false`, то детально разбирать, собственно, и нечего. Зато это прекрасный повод, что бы ещё раз повторить определение. Каждый раз, когда встречается часть кода, работа которой зависит от заранее определенного значения-константы, стоит подумать, можно ли ограничивать тип литеральным типом, сможет ли это повысить типобезопасность и семантику кода.

ts

```ts
function setFlag(flag: true | "true"): void {

}
```

Примитивный литеральный тип `boolean` является уникальным для *TypeScript*, в *JavaScript* подобного типа не существует.

[18.4] Литеральный Тип Unique Symbol (unique symbol) уникальный символьный тип

Несмотря на то, что тип данных `symbol` является уникальным для программы, с точки зрения системы типов, он не может гарантировать типобезопасность.

ts

```ts
function f(key: symbol) {
  // для успешного выполнения программы предполагается, что параметр
key будет принадлежать к типу Symbol.for('key')...
}
```

```
f(Symbol.for('bad key')); // ... тем не менее функцию f() можно вызвать
с любым другим символом
```

Для того, что бы избежать подобного сценария, *TypeScript* добавил новый примитивный литеральный тип `unique symbol`. `unique symbol` является подтипом `symbol` и указывается в аннотации с помощью литерального представления `unique symbol`.

Экземпляр `unique symbol` создается теми же способами, что и обычный `symbol` при помощи прямого вызова конструктора `Symbol()` или статического метода `Symbol.for()`. Но, в отличие от обычного `symbol`, `unique symbol` может быть указан только в аннотации константы (`const`) и поля класса (`static`) объявленного с модификатором `readonly`.

ts

```ts
const v0: unique symbol = Symbol.for('key'); // Ok
let v1: unique symbol = Symbol.for('key'); // Error
var v2: unique symbol = Symbol.for('key'); // Error

class Identifier {
    public static readonly f0: unique symbol = Symbol.for('key'); // Ok

    public static f1: unique symbol = Symbol.for('key'); // Error
    public f2: unique symbol = Symbol.for('key'); // Error
}
```

Кроме того, что бы ограничить значение до значения принадлежащего к типу `unique symbol`, требуется прибегать к механизму запроса типа, который подробно был рассмотрен в главе “Типы - Type Queries (запросы типа), Alias (псевдонимы типа)”.

ts

```ts
const KEY: unique symbol = Symbol.for('key');

// аннотация параметра и возвращаемого из функции типа при помощи
механизма запросов типа
function f(key: typeof KEY): typeof KEY {
    return key;
}

f(KEY); // Ok
f(Symbol('key')); // Error
f(Symbol.for('key')); // Error
```

Поскольку каждый `unique symbol` имеет собственное представление в системе типов, совместимыми могут считаться только символы, имеющие идентичную ссылку на объявление.

ts

```
const KEY: unique symbol = Symbol.for('key');
const OTHER_KEY: unique symbol = Symbol.for('key');

if (KEY === OTHER_KEY) { // Ошибка, unique symbol не равно unique symbol

}

function f(key: typeof KEY): typeof KEY {
    return key;
}

let key = KEY; // let key: symbol; // symbol !== unique symbol

f(key); // Error
```

Тип `unique symbol` предназначен для аннотирования уникальных символьных литералов. С его помощью реализуется задуманное для *JavaScript* поведение в типизированной среде *TypeScript*.

[18.5] Литеральный тип Enum (Enum Literal Types)

Литеральный тип `enum` ограничивается литеральными значениями его констант. Это утверждение верно с одной оговоркой: правило совместимости типов для перечисления, у которого имеются константы с числовым значением, распространяется и на литеральный тип `enum` .

Напомним, если перечисление составляют только строковые константы, то в качестве значения может быть присвоено только константы перечисления.

ts

```
enum Berries {
    Strawberry = "strawberry",
    Raspberry = "raspberry",
    Blueberry = "blueberry"
}

type RedBerry = Berries.Raspberry | Berries.Strawberry;

var berry: RedBerry = Berries.Strawberry; // Ok
var berry: RedBerry = Berries.Raspberry; // Ok
var berry: RedBerry = Berries.Blueberry; // Error
```

```
var berry: RedBerry = 123; // Error
var berry: RedBerry = "strawberry"; // Error
```

В том же случае, если в перечислении присутствует константа с числовым значением, в качестве значения может быть присвоено любое число.

ts

```ts
enum Fruits {
    Apple,
    Pear,
    Banana = "banana"
}

type FruitGrowOnTree = Fruits.Apple | Fruits.Pear;

var fruit: FruitGrowOnTree = Fruits.Apple; // Ok
var fruit: FruitGrowOnTree = Fruits.Pear; // Ok
var fruit: FruitGrowOnTree = Fruits.Banana; // Error
var fruit: FruitGrowOnTree = 123; // Ok!
var fruit: FruitGrowOnTree = "apple"; // Error
```

Правила литеральных типов `enum` распространяются и на перечисления объявленных с помощью ключевого слова `const`.

Примитивный литеральный тип `enum` является уникальным для *TypeScript*, в *JavaScript* подобного типа не существует.

Глава 19

Object, Array, Tuple

Пришло время рассмотреть такие типы данных как `Object` и `Array`, с которыми разработчики *JavaScript* уже хорошо знакомы. А также неизвестный им тип данных `Tuple`, который, как мы скоро убедимся, не представляет собой ничего сложного.

[19.0] Object (object) — ссылочный объектный тип

Ссылочный тип данных `Object` является базовым для всех ссылочных типов в *TypeScript*.

Помимо того, что в *TypeScript* существует объектный тип `Object`, представляющий одноименный конструктор из *JavaScript*, также существует тип `object`, представляющий любое объектное значение. Поведение типа указанного с помощью ключевого слова `object` и интерфейса `Object` различаются.

Переменные, которым указан тип с помощью ключевого слова `object`, не могут хранить значения примитивных типов, чьи идентификаторы (имена) начинаются со строчной буквы (`number`, `string` и т.д.). В отличие от них тип интерфейс `Object` совместим с любым типом данных.

ts

```ts
let o: object;
let O: Object;

o = 5; // Error
```

```
O = 5; // Ok

o = ''; // Error
O = ''; // Ok

o = true; // Error
O = true; // Ok

o = null; // Error, strictNullChecks = true
O = null; // Error, strictNullChecks = true

o = undefined; // Error, strictNullChecks = true
O = undefined; // Error, strictNullChecks = true
```

По факту, тип, указанный как `object`, соответствует чистому объекту, то есть не имеющему никаких признаков (даже унаследованных от типа `Object`). В, то время как значение, ограниченное типом `Object`, будет включать все его признаки (методы `hasOwnProperty()` и т.п.). При попытке обратиться к членам объекта, не задекларированным в интерфейсе `Object`, возникнет ошибка. Напомним, что в случаях, когда тип нужно сократить до базового, сохранив при этом возможность обращения к специфичным (определенным пользователем) членам объекта, нужно использовать тип `any`.

ts

```
class SeaLion {
    rotate(): void {}

    voice(): void {}
}

let seaLionAsObject: object = new SeaLion(); // Ok
seaLionAsObject.voice(); // Error

let seaLionAsAny: any = new SeaLion(); // Ok
seaLionAsAny.voice(); // Ok
```

Тип интерфейса `Object` идентичен по своей работе одноименному типу из *JavaScript*. Несмотря на то, что тип указанный с помощью ключевого слова `object` имеет схожее название, его поведение отличается от типа интерфейса.

[19.1] Array (type[]) ссылочный массивоподобный тип

Ссылочный тип данных `Array` является типизированным спископодобным объектом, содержащим логику для работы с элементами.

Тип данных `Array` указывается с помощью литерала массива, перед которым указывается тип данных `type[]` .

Если при объявлении массива указать тип `string[]` , то он сможет хранить только элементы принадлежащие или совместимые с типом `string` (например `null` , `undefined` , `literal type string`).

ts

```ts
var animalAll: string[] = [
    'Elephant',
    'Rhino',
    'Gorilla'
];

animalAll.push(5); // Error
animalAll.push(true); // Error
animalAll.push(null); // Ok
animalAll.push(undefined); // Ok
```

В случае неявного указания типа вывод типов самостоятельно укажет тип как `string[]` .

ts

```ts
var animalAll = [
    'Elephant',
    'Rhino',
    'Gorilla'
]; // animalAll : string[]
```

Если требуется, что бы массив хранил смешанные типы данных, то один из способов это сделать — указать тип объединение (`Union`). Нужно обратить внимание на то, как трактуется тип данных `Union` при указании его массиву. Может показаться, что указав в качестве типа тип объединение `Union` , массив `(Elephant | Rhino | Gorilla)[]` может состоять только из какого-то одного перечисленного типа `Elephant` , `Rhino`

или `Gorilla`. Но это не совсем так. Правильная трактовка гласит, что каждый элемент массива может принадлежать к типу `Elephant` или `Rhino`, или `Gorilla`. Другими словами, типом, к которому принадлежит массив, ограничивается не весь массив целиком, а каждый отдельно взятый его элемент.

ts

```ts
class Elephant {}
class Rhino {}
class Gorilla {}

var animalAll: (Elephant | Rhino | Gorilla)[] = [
    new Elephant(),
    new Rhino(),
    new Gorilla()
];
```

Если для смешанного массива не указать тип явно, то вывод типов самостоятельно укажет все типы, которые хранятся в массиве. Более подробно эта тема будет рассмотрена в главе “Типизация - Вывод типов”.

В случае, если при создании экземпляра массива типы его элементов неизвестны, то следует указать в качестве типа тип `any`.

ts

```ts
let dataAll: any[] = [];

dataAll.push(5); // Ok -> number
dataAll.push('5'); // Ok -> string
dataAll.push(true); // Ok -> boolean
```

Нужно стараться как можно реже использовать массивы со смешанными типами, а к массивам с типом `any` нужно прибегать только в самых крайних случаях. Кроме того, как было рассказано в главе “Экскурс в типизацию - Совместимость типов на основе вариантности”, нужно крайне осторожно относиться к массивам, у которых входные типы являются ковариантными.

В случаях, требующих создания экземпляра массива с помощью оператора `new`, необходимо прибегать к типу глобального обобщённого интерфейса `Array<T>`. Обобщения будут рассмотрены чуть позднее, а пока нужно запомнить следующее. При попытке создать экземпляр массива путем вызова конструктора, операция завершится успехом в тех случаях, когда создаваемый массив будет инициализирован пустым либо с элементами одного типа данных. В случаях смешанного массива его тип необходимо конкретизировать явно с помощью параметра типа заключенного в угловые скобки. Если сейчас это не понятно, не переживайте, в будущем это будет рассмотрено очень подробно.

ts

```
let animalData: string[] = new Array(); //Ok
let elephantData: string[] = new Array('Dambo'); // Ok
let lionData: (string | number)[];

lionData = new Array('Simba', 1); // Error
lionData = new Array('Simba'); // Ok
lionData = new Array(1); // Ok
let deerData: (string | number)[] = new Array<string | number>('Bambi',
1); // Ok
```

В *TypeScript* поведение типа `Array<T>` идентично поведению одноимённого типа из *JavaScript*.

[19.2] Tuple ([T0, T1, …, Tn]) тип кортеж

Тип `Tuple` (кортеж) описывает строгую последовательность множества типов, каждый из которых ограничивает элемент массива с аналогичным индексом. Простыми словами кортеж задает уникальный тип для каждого элемента массива. Перечисляемые типы обрамляются в квадратные скобки, а их индексация, так же как у массива начинается с нуля - `[T1, T2, T3]` . Типы элементов массива, выступающего в качестве значения, должны быть совместимы с типами обусловленных кортежем под аналогичными индексами.

Другими словами, если кортеж составляет последовательность типов `string` и `numb er` , то в качестве значения должен выступать массив, первый элемент которого совместим с типом `string` , а второй с `number` . В иных ситуациях неизбежно возникнет ошибка.

ts

```
let v0: [string, number] = ['Dambo', 1]; // Ok
let v1: [string, number] = [null, undefined]; // Error -> null не
string, а undefined не number
let v3: [string, number] = [1, 'Simba']; // Error -> порядок обязателен
let v4: [string, number] = [,, ]; // Error -> пустые элементы массива
приравниваются к undefined
```

Длина массива-значения должна соответствовать количеству типов, указанных в `Tuple` .

ts

```
let elephantData: [string, number] = ['Dambo', 1]; // Ok
let lionData: [string, number] = ['Simba', 1, 1]; // Error, лишний
элемент
let fawnData: [string, number] = ['Bambi']; // Error, не достает одного
элемента
let giraffeData: [string, number] = []; // Error, не достает всех
элементов
```

Но это правило не мешает добавить новые элементы после того, как массив был присвоен ссылке (ассоциирован со ссылкой). Но элементы, чьи индексы выходят за пределы установленные кортежем, обязаны иметь тип, совместимый с одним из перечисленных в этом кортеже.

ts

```
let elephantData: [string, number] = ['Dambo', 1];
elephantData.push(1941); // Ok
elephantData.push('Disney'); // Ok
elephantData.push(true); // Error, тип boolean, в, то время, как
допустимы только типы совместимые с типами string и number

elephantData[10] = ''; // Ok
elephantData[11] = 0; // Ok

elephantData[0] = ''; // Ok, значение совместимо с типом заданном в
кортеже
elephantData[0] = 0; // Error, значение не совместимо с типом заданном
в кортеже
```

Массив, который связан с типом кортежем, ничем не отличается от обычного, за исключением способа определения типа его элементов. При попытке присвоить элемент под индексом 0 переменной с типом `string`, а элемент под индексом `1` переменной с типом `number`, операции присваивания завершатся успехом. Но, несмотря на то, что элемент под индексом `2` хранит значение, принадлежащее к типу `string`, оно не будет совместимо со `string`. Дело в том, что элементы, чьи индексы выходят за пределы установленные кортежем, принадлежат к типу объединению (`Union`). Это означает, что элемент под индексом `2` принадлежит к типу `string | number`, а это не, то же самое, что тип `string`.

ts

```
let elephantData: [string, number] = ['Dambo', 1]; // Ok

elephantData[2] = 'nuts';

let elephantName: string = elephantData[0]; // Ok, тип string
let elephantAge: number = elephantData[1]; // Ok, тип number
let elephantDiet: string = elephantData[2]; // Error, тип string |
number
```

Есть два варианта решения этой проблемы. Первый вариант, изменить тип переменной со `string` на тип объединение `string | number`, что ненадолго избавит от проблемы совместимости типов. Второй, более подходящий вариант, прибегнуть к приведению типов, который детально будет рассмотрен позднее.

В случае, если описание кортежа может навредить семантике кода, его можно поместить в описание псевдонима типа (`type`).

ts

```ts
type Tuple = [number, string, boolean, number, string];

let v1: [number, string, boolean, number, string]; // плохо
let v2: Tuple; // хорошо
```

Кроме того, тип кортеж можно указывать в аннотации остаточных параметров (`...rest`).

ts

```ts
function f(...rest: [number, string, boolean]): void {}

let tuple: [number, string, boolean] = [5, '', true];
let array = [5, '', true];

f(5); // Error
f(5, ''); // Error
f(5, '', true); // Ok
f(...tuple); // Ok
f(tuple[0], tuple[1], tuple[2]); // Ok
f(...array); // Error
f(array[0], array[1], array[2]); // Error, все элементы массива
принадлежат к типу string | number | boolean, в, то время как первый
элемент кортежа принадлежит к типу number
```

Помимо этого, типы, указанные в кортеже, могут быть помечены как необязательные с помощью необязательного модификатора `?` .

ts

```ts
function f(...rest: [number, string?, boolean?]): void {}

f(); // Error
f(5); // Ok
f(5, ''); // Ok
f(5, '', true); // Ok
```

У кортежа, который включает типы помеченные как не обязательные, свойство длины принадлежит к типу объединения (`Union`), состоящего из литеральных числовых типов.

ts

```ts
function f(...rest: [number, string?, boolean?]): [number, string?,
boolean?] {
    return rest;
}

let l = f(5).length; // let l: 1 | 2 | 3
```

Кроме того, для кортежа применим механизм распространения (`spread`), который может быть указан в любой части определения типа. Но существуют два исключения. Во-первых, определение типа кортежа может включать только одно распространение.

ts

```ts
/**
 * [0] A rest element cannot follow another rest element.ts(1265)
 *
 */
let v0: [...boolean[], ...string[]]; // Error [0]
let v1: [...boolean[], boolean, ...string[]]; // Error [0]

let v2: [...boolean[], number]; // Ok
let v3: [number, ...boolean[]]; // Ok
let v4: [number, ...boolean[], number]; // Ok
```

И во вторых, распространение не может быть указанно перед необязательными типами.

ts

```ts
/**
 * [0] An optional element cannot follow a rest element.ts(1266)
 *
 */
let v5: [...boolean[], boolean?]; // Error [1]

let v6: [boolean?, ...boolean[]]; // Ok
```

В результате распространения, получается тип с логически предсказуемой последовательностью типов, определяющих кортеж.

ts

```ts
type Strings = [string, string];
type BooleanArray = boolean[];

// type Unbounded0 = [string, string, ...boolean[], symbol]
type Unbounded0 = [...Strings, ...BooleanArray, symbol];

// type Unbounded1 = [string, string, ...boolean[], symbol, string,
```

```
string]
type Unbounded1 = [ ...Strings, ...BooleanArray, symbol, ...Strings]
```

Стоит заметить, что поскольку механизм распространения участвует в рекурсивном процессе формирования типа, способного значительно замедлять компилятор, установленно ограничение в размере 10000 итераций.

Механизм объявления множественного распространения (`spread`) значительно упрощает аннотирование сигнатуры функции при реализации непростых сценариев, один из которых будет рассмотрен далее в главе (Массивоподобные readonly типы)[].

Еще несколько неочевидных моментов в логике кортежа связанны с выводом типов и будут рассмотрены в главе “Типизация - Вывод типов” (*см реализацию функции concat*).

Помимо этого семантику типов кортежей можно повышать за счет добавления им меток.

ts

```ts
// пример безликого кортежа

const f = (p: [string, number]) => {}

/**
 * автодополнение -> f(p: [string, number]): void
 *
 * Совершенно не понятно чем конкретно являются
 * элементы представляемые типами string и number
 */
f0()
```

ts

```ts
// пример кортежа с помеченными элементами

const f = (p: [a: string, b: number]) => {};

/**
 * автодополнение -> f(p: [a: string, b: number]): void
 *
 * Теперь мы знаем, что функция ожидает не просто
 * строку и число, а аргумент "a" и аргумент "b",
 * которые в реальном проекте будут иметь более
 * осмысленное смысловое значение, например "name" и "age".
 */
f1()
```

Поскольку метки являются исключительной частью синтаксиса *TypeScript* они не имеют никакой силы в коде при деструктуризации массива, представленного типом кортежа.

ts

```
const f = (p: [a: string, b: number]) => {
    let [c, d] = p;
};
```

Единственное правило, касающееся данного механизма, заключается в том, что кортеж, содержащий метки, не может содержать элементы описанные только типами.

ts
```
type T = [a: number, b: string, boolean]; // Error -> Tuple members
must all have names or all not have names.ts(5084)
```

Напоследок стоит обратить внимание на тот факт, что тип переменной при присвоении ей инициализированного массива без явного указания типа, будет выведен как массив. Другими словами, вывод типа неспособен вывести тип кортеж.

ts
```
let elephantData = ['Dambo', 1]; // type Array (string | number)[]
```

Тип `Tuple` является уникальным для *TypeScript*, в *JavaScript* подобного типа не существует.

Глава 20

Function, Functional Types

Функция — это ключевая концепция *JavaScript*. Функции присваиваются в качестве значений переменным и передаются как аргументы при вызове других функций. Поэтому не удивительно, что *TypeScript* очень много внимания уделяет возможностям *функционального типа*, к которым, начиная с текущей главы, повествование периодически будет возвращаться.

[20.0] Function Types - тип функция

В *TypeScript* тип `Function` представляет одноименный *JavaScript* конструктор, являющийся базовым для всех функций. Тип `Function` можно указывать в аннотации типа тогда, когда о сигнатуре функции ничего не известно или в качестве значения могут выступать функции с несовместимыми сигнатурами.

ts

```
function f1(p1: number): string {
    return p1.toString();
}

function f2(p1: string): number {
    return p1.length;
}

let v1: Function = f1;
let v2: Function = f2;
```

При этом нельзя забывать, что по канонам статически типизированных языков, архитектуру программы нужно продумывать так, что бы сводить присутствие высших в иерархии типов к нулю. В тех случаях, когда сигнатура функции известна, тип стоит конкретизировать при помощи определения более конкретных функциональных типов.

Поведение типа `Function` идентично одноимённому типу из *JavaScript*.

[20.1] Functional Types - функциональный тип

Помимо того, что в *TypeScript* существует объектный тип `Function`, также существует функциональный тип, с помощью которого осуществляется описание сигнатур функциональных выражений.

Функциональный тип обозначается с помощью пары круглых скобок `()`, после которых располагается стрелка, а после неё обязательно указывается тип возвращаемого значения `() => type`. При наличии у функционального выражения параметров, их декларация заключается между круглых скобок `(p1: type, p2: type) => type`.

ts

```
type FunctionalType = (p1: type, p2: type) => type;
```

Если декларация сигнатуры функционального выражения известна, то рекомендуется использовать более конкретный функциональный тип, поскольку он в большей степени соответствует типизированной атмосфере.

ts

```
type SumFunction = (a: number, b: number) => number;

const sum: SumFunction = (a: number, b: number): number => a + b;
```

Поведение функционального типа указывающегося с помощью функционального литерала идентично поведению типа `Function`, но при этом оно более конкретно и поэтому предпочтительнее.

[20.2] this в сигнатуре функции

Ни для кого не будет секретом, что в *JavaScript* при вызове функций можно указать их контекст. В львиной доле случаев, возможность изменять контекст вызова функции является нежелательным поведением *JavaScript*, но только не в случае реализации конструкции, называемой *функциональная примесь* (functional mixins).

Функциональная примесь — это функция, в теле которой происходит обращение к членам, объявленных в объекте, к которому она *“примешивается”*. Проблем не возникнет, если подобный механизм реализуется в динамически типизированном языке, каким является *JavaScript*.

ts

```
// .js

class Animal {
    constructor(){
        this.type = 'animal';
    }
}

function getType() {
    return this.type;
}

let animal = new Animal();
animal[getType.name] = getType;

console.log(animal.getType()); // animal
```

Но в статически типизированном языке такое поведение должно быть расценено как ошибочное, поскольку у функции нет присущего объектам признака `this`. Несмотря на это в *JavaScript*, а значит и в *TypeScript*, контекст самой программы (или, по другому,

глобальный объект) является объектом. Это, в свою очередь, означает, что не существует места, в котором бы ключевое слово `this` привело к возникновению ошибки (для запрещения `this` в нежелательных местах нужно активировать опцию компилятора `--noImplicitThis`). Но при этом за невозможностью предугадать поведение разработчика, в *TypeScript* ссылка `this` вне конкретного объекта ссылается на тип `any`, что лишает ide автодополнения. Для таких и не только случаев была реализованна возможность декларировать тип `this` непосредственно в функциях.

`this` указывается в качестве первого параметра любой функции и как обычный параметр имеет аннотацию типа, устанавливающую принадлежность к конкретному типу.

ts

```ts
interface IT1 {p1: string;}

function f1(this: IT1): void {}
```

Несмотря на то, что `this` декларируется в параметрах функции, таковым оно не считается. Поведение функции с декларацией `this` аналогично поведению функции без декларации `this`. Единственное на, что стоит обратить внимание, что в случае указания принадлежности к типу отличного от `void`, не получится вызвать функцию вне указанного контекста.

ts

```ts
interface IT1 { p1: string; }

function f1(this: void): void {}
function f2(this: IT1): void {}
function f3(): void {}

f1(); // Ok
f2(); // Error
f3(); // Ok

let v1 = { // v1: {f2: (this: IT1) => void;}
    f2: f2
};

v1.f2(); // Error

let v2 = { // v2: {p1: string; f2: (this: IT1) => void;}
    p1: '',
    f2: f2
};

v2.f2(); // Ok
```

Кроме того, возможность ограничивать поведение ключевого слова `this` в теле функции призвано частично решить самую часто возникающую проблему, связанную с

потерей контекста. Вряд ли найдется разработчик *JavaScript*, который может похвастаться, что ни разу не сталкивался с потерей контекста при передаче метода объекта в качестве функции обратного вызова (*callback*). В случаях, когда в теле метода происходит обращение через ссылку `this` к членам объекта, в котором он определен, то при потере контекста, в лучшем случае возникнет ошибка. В худшем, предполагающем, что в новом контексте будут присутствовать схожие признаки, возникнет трудно выявляемая ошибка.

ts

```ts
class Point {
    constructor(
        public x: number = 0,
        public y: number = 0
    ){}
}

class Animal {
    private readonly position: Point = new Point();

    public move({clientX, clientY}: MouseEvent): void {
        this.position.x = clientX;
        this.position.y = clientY;
    }
}

let animal = new Animal();

document.addEventListener('mousemove', animal.move); // ошибка во время
выполнения
```

Для этих случаев *TypeScript* предлагает ограничить ссылку на контекст с помощью конкретизации типа ссылки `this` .

Так как реальный пример, иллюстрирующий полную картину, получается очень объемным, то ограничимся одним методом, реализующим обсуждаемое поведение.

ts

```
type IContextHandler = (this: void, event: MouseEvent) => void;

class Controller {
    public addEventListener(type: string, handler: IContextHandler):
void {}
}


let animal = new Animal();
let controller = new Controller();

controller.addEventListener('mousemove', animal.move); // ошибка во
время выполнения
```

Стоит заметить, что одной конкретизации типа ссылки `this` в слушателе событий недостаточно. Для того, что бы пример заработал должным образом, необходимо конкретизировать ссылку `this` в самом слушателе событий.

ts

```
class Point {
    constructor(
        public x: number = 0,
        public y: number = 0
    ){}
}

class Animal {
    private readonly position: Point = new Point();

    public move(this: Animal, {clientX, clientY}: MouseEvent): void
{ // <= изменения
        this.position.x = clientX;
        this.position.y = clientY;
    }
}


type IContextHandler = (this: void, event: MouseEvent) => void;

class Controller {
    public addEventListener(type: string, handler: IContextHandler):
void {}
}


let animal = new Animal();
let controller = new Controller();

controller.addEventListener('mousemove', animal.move); // ошибка во
время компиляции
```

```
controller.addEventListener('mousemove', event =>
animal.move(event)); // Ok
```

Также стоит обратить внимание на одну неочевидную на первый взгляд деталь. Когда мы передаем слушатель, обернув его в стрелочную функцию либо в метод функции `.bind`, ошибки не возникает только потому, что у передаваемой функции отсутствует декларация `this`.

Глава 21

Interfaces

Несмотря на то, что тема относящаяся к интерфейсам очень проста, именно она вызывает наибольшее количество вопросов у начинающих разработчиков. Поэтому, такие вопросы как *для чего нужны интерфейсы*, *когда их применять, а когда нет*, будут подробно рассмотрены в этой главе.

[21.00] Общая теория

По факту, интерфейс затрагивает сразу несколько аспектов создания программ, относящихся к проектированию, реализации, конечной сборке. Поэтому, что бы понять предназначение интерфейса, необходимо рассмотреть каждый аспект по отдельности.

Первый аспект — реализация — предлагает рассматривать создаваемые экземпляры как социальные объекты, чья публичная часть инфраструктуры была оговорена в контракте, к коему относится интерфейс. Другими словами, интерфейс — это контракт, реализация которого гарантирует наличие оговоренных в нем членов потребителю экземпляра. Поскольку интерфейс описывает исключительно типы членов объекта (поля, свойства, сигнатуры методов), они не могут гарантировать, что сопряженная с ними логика будет соответствовать каким-либо критериям. По этому случаю была принята методология называемая *контрактное программирование*. Несмотря на то, что данная методология вызывает непонимание у большинства начинающих разработчиков, в действительности она очень проста. За этим таинственным термином скрываются рекомендации придерживаться устной или письменной спецификации при реализации логики, сопряженной с оговоренными в интерфейсе членами.

Второй аспект — проектирование — предлагает проектировать объекты менее независимыми за счет отказа от конкретных типов (классов) в пользу интерфейсов. Ведь пока тип переменной или параметра представляется классовым типом, невозможно будет присвоить значение соответствующее этому типу, но не совместимое с ним. Под соответствующим подразумевается соответствие по всем обязательным признакам, но не состоящим в отношениях наследования. И хотя в *TypeScript* из-за реализации *номинативной типизации* подобной проблемы не существует, по возможности рекомендуется придерживаться классических взглядов.

Третий аспект — сборка — вытекает из второго и предполагает уменьшение размера компилируемого пакета (*bundle*) за счет отказа от конкретных типов (классов). Фактически, если какой-либо объект требуется пакету лишь для выполнения операций над ним, последнему вовсе не нужно содержать определение первого. Другими словами, скомпилированный пакет не должен включать определение класса со всей его логикой только потому, что он указан в качестве типа. Для этого как нельзя лучше подходят типы, представленные интерфейсами. Хотя нельзя не упомянуть, что данная проблема не имеет никакого практического отношения к разработчикам на языке *TypeScript*, поскольку его (или точнее сказать *JavaScript*) модульная система лишена подобного недостатка.

Вот эти несколько строк, описывающие оговоренные в самом начале аспекты, заключают в себе ответы на все возможные вопросы, которые только могут возникнуть относительно темы сопряженной с интерфейсами. Если ещё более доступно, то интерфейсы нужны для снижения зависимости и наложения обязательств на реализующие их классы. Интерфейсы стоит применять всегда и везде, где это возможно. Это не только повысит семантическую привлекательность кода, но и сделает его более поддерживаемым.

Не лишним будет добавить, что интерфейсы являются фундаментальной составляющей идеологии как типизированных языков, так и объектно-ориентированного программирования.

Такая известная группа программистов, как *“Банда четырех”* (*Gang of Four*, сокращённо *GoF*), в своей книге, положившей начало популяризации шаблонов проектирования, описывали интерфейс как ключевую концепцию *объектно-ориентированного программирования* (*ооп*). Понятие интерфейса является настолько важным, что в книге был сформулирован принцип объектно-ориентированного проектирования, который звучит так: *Программируйте в соответствии с интерфейсом, а не с реализацией*.

Другими словами, авторы советуют создавать систему, которой вообще ничего не будет известно о реализации. Проще говоря, создаваемая система должна быть построена на типах, определяемых интерфейсами, а не на типах, определяемых классами.

С теорией закончено. Осталось подробно рассмотреть реализацию интерфейсов в *TypeScript*.

[21.01] Интерфейс в TypeScript

TypeScript предлагает новый тип данных, определяемый с помощью синтаксической конструкции называемой *интерфейс* (`interface`).

Interface — это синтаксическая конструкция, предназначенная для описания открытой (`public`) части объекта без реализации (*api*). Хотя не будет лишним упомянуть, что существуют языки позволяющие реализовывать в интерфейсах поведение, рассматриваемое как поведение по умолчанию.

Класс, реализующий интерфейс, обязан реализовать все описанные в нём члены. Поэтому интерфейс является гарантией наличия описанных в нем характеристик у реализующего его объекта. Все члены, описанные в интерфейсе, неявно имеют модификатор доступа `public` . Интерфейс предназначен для описания *api* или другими словами состояния и поведения предназначенного для взаимодействия внешнего мира с объектом.

[21.02] Объявление (declaration)

В *TypeScript* интерфейс объявляется с помощью ключевого слова `interface` , после которого указывается идентификатор (имя), за которым следует заключенное в фигурные скобки тело содержащее описание.

ts

```ts
interface Identifier {
    // тело интерфейса
}
```

Объявление интерфейса возможно как в контексте модуля, так и в контексте функции или метода.

ts

```
interface Identifier {} // контекст модуля

class T {
    public method(): void {
        interface Identifier {} // контекст метода
    }
}

function func(): void {
   interface Identifier {} // контекст функции
}
```

[21.03] Конвенции именования интерфейсов

Прежде чем продолжить, нужно обратить внимание на такой аспект, как конвенции именования интерфейсов. Существует два вида именования.

Первый вид конвенций родом из языка *Java* — они предлагают именовать интерфейсы точно так же как и классы. Допускаются имена прилагательные.

ts

```
interface Identifier {}
```

Второй вид предлагает использовать конвенции языка *C#*, по которым интерфейсы именуются так же как классы, но с префиксом `I`, что является сокращением от *Interface*. Такой вид записи получил название *“венгерская нотация”* в честь программиста венгерского происхождения, работавшего в компании *MicroSoft*. Допускаются имена прилагательные.

ts

```
interface IIdentifier {}
```

Чтобы сразу расставить все точки над i, стоит заметить, что в дальнейшем идентификаторы интерфейсов будут указываться по конвенциям *C#*.

[21.04] Реализация интерфейса (implements)

Как уже было сказано в самом начале, все члены интерфейса являются открытыми (`public`) и их объявление не может содержать модификатор `static` . Кроме того, в *TypeScript* интерфейсы не могут содержать реализацию.

Класс, реализующий интерфейс, обязан реализовывать его в полной мере. Любой класс, который хочет реализовать интерфейс, должен указать это с помощью ключевого слова `implements` , после которого следует идентификатор реализуемого интерфейса. Указание реализации классом интерфейса располагается между идентификатором класса и его телом.

ts

```ts
interface IAnimal {
    nickname: string;

    execute(command: string): void;
}

class Bird implements IAnimal {
    nickname: string;

    execute(command: string): void {}
}
```

Один класс может реализовывать сколько угодно интерфейсов. В этом случае реализуемые интерфейсы должны быть перечислены через запятую.

ts

```ts
interface IAnimal {}
interface IOviparous {} // указывает на возможность откладывать яйца

class Bird implements IAnimal, IOviparous {}
```

В случае, когда класс расширяет другой класс, указание реализации (`implements`) следует после указания расширения (`extends`).

ts

```ts
interface IAnimal {}
interface IOviparous {}
```

```
class Bird implements IAnimal, IOviparous {}

interface IFlyable {}

class Eagle extends Bird implements IFlyable {}
```

[21.05] Декларация свойств get и set (accessors)

Несмотря на то, что в интерфейсе можно декларировать поля и методы, в нем нельзя декларировать свойства `get` и `set` (*аксессоры*). Но, несмотря на это, задекларированное в интерфейсе поле может быть совместимо не только с полем, но и аксессорами. При этом нет разницы, будет в объекте объявлен *getter*, *setter* или оба одновременно.

ts

```
interface IAnimal {
    id: string;
}

// только get
class Bird implements IAnimal {
    get id(): string {
        return 'bird';
    }
}

// только set
class Fish implements IAnimal {
    set id(value: string) {}
}

// и get и set
class Insect implements IAnimal {
    get id(): string {
     return 'insect';
    }

    set id(value: string) {}
}
```

[21.06] Указание интерфейса в качестве типа (interface types)

Класс, реализующий интерфейс, принадлежит к типу этого интерфейса. Класс, унаследованный от класса реализующего интерфейс, также наследует принадлежность к реализуемым им интерфейсам. В подобных сценариях говорят, что класс наследует интерфейс.

ts

```ts
interface IAnimal {}

class Bird implements IAnimal {}

class Raven extends Bird {}

let bird: IAnimal = new Bird();
let raven: IAnimal = new Raven();
```

Класс, реализующий множество интерфейсов, принадлежит к типу каждого из них. Когда экземпляр класса, реализующего интерфейс, присваивают ссылке с типом интерфейса, то говорят, что экземпляр был *ограничен* типом интерфейса. То есть, функциональность экземпляра класса урезается до описанного в интерфейсе (подробнее об этом речь пойдет в главе “Типизация - Совместимость объектов” и “Типизация - Совместимость функций”).

ts

```ts
interface IAnimal {
    name: string;
}

interface IFlyable {
    flightHeight: number;
}

interface IIdentifiable {
    id: string;
}

class Animal implements IAnimal {
    constructor(readonly name: string) {}
}
```

```
class Bird extends Animal implements IFlyable {
    public flightHeight: number = 500;
}

var animal: IAnimal = new Bird('bird'); // экземпляр Bird ограничен до
типа IAnimal
var fly: IFlyable = new Bird('bird'); // экземпляр Bird ограничен до
типа IFlyable
```

Несмотря на то, что интерфейс является синтаксической конструкцией и может указываться в качестве типа, после компиляции от него не остается и следа. Это, в свою очередь, означает, что интерфейс, как тип данных, может использоваться только на этапе компиляции. Другими словами, компилятор сможет предупредить об ошибках несоответствия объекта описанному интерфейсу, но проверить на принадлежность к типу интерфейса с помощью операторов `typeof` или `instanceof` не получится, поскольку они выполняются во время выполнения программы. Но в *TypeScript* существует механизм (который будет рассмотрен далее в главе “Типизация - Защитники типа”), позволяющий в некоторой мере решить эту проблему.

[21.07] Расширение интерфейсов (extends interface)

Если множество логически связанных интерфейсов требуется объединить в один тип, то нужно воспользоваться механизмом расширения интерфейсов. Наследование интерфейсов осуществляется с помощью ключевого слова `extends`, после которого через запятую идет один или несколько идентификаторов расширяемых интерфейсов.

ts

```
interface IIdentifiable {}
interface ILiving {}

// интерфейсы IIdentifiable и ILiving вместе образуют логически
связанную композицию,
// которую можно выделить в интерфейс IAnimal
interface IAnimal extends IIdentifiable, ILiving {}
```

Для тех, кто только знакомится с понятием интерфейса, будет не лишним узнать о *“Принципе разделения интерфейсов”* (*Interface Segregation Principle* или сокращенно *ISP*), который гласит, что более крупные интерфейсы нужно *“дробить”* на более мелкие

интерфейсы. Но нужно понимать, что условия дробления диктуются конкретным приложением. Если во всех случаях руководствоваться только принципами, то можно раздуть небольшое приложение до масштабов вселенной.

Для примера представьте приложение, которое только выводит в консоль информацию о животных. Так как над объектом `Animal` будет выполняться только одна операция, то можно не бояться разгневать богов объектно-ориентированного проектирования и включить все нужные характеристики прямо в интерфейс `IAnimal`.

ts

```ts
interface IAnimal {
    id: string;
    age: number;
}

class Animal implements IAnimal {
    public age: number = 0;

    constructor(readonly id: string) {}
}

class AnimalUtil {
    public static print(animal: IAnimal): void {
        console.log(animal);
    }
}

class Bird extends Animal {}

class Raven extends Bird {
    constructor() {
        super('raven');
    }
}

let raven: Raven = new Raven();

AnimalUtil.print(raven);
```

В такой программе, кроме достоинства архитектора, ничего пострадать не может, так как она выполняет только одну операцию вывода информации о животном.

Но, если переписать программу, что бы она выполняла несколько не связанных логически операций над одним типом, в данном случае `IAnimal`, то ситуация изменится на противоположную.

ts

```ts
interface IAnimal {/*...*/}

class Animal implements IAnimal {/*...*/}
```

```
class AnimalUtil {
    public static printId(animal: IAnimal): void {
        console.log(animal.id); // вывод  id
    }

    public static printAge(animal: IAnimal): void {
        console.log(animal.age); // вывод age
    }
}

class Bird extends Animal {}
class Raven extends Bird {/*...*/}

let raven: Raven = new Raven();

AnimalUtil.printId(raven);
AnimalUtil.printAge(raven);
```

В этом случае программа нарушает принцип *ISP*, так как статические методы `printId` и `printAge` получили доступ к данным, которые им не требуются для успешного выполнения. Это может привести к намеренной или случайной порче данных.

ts

```
class AnimalUtil {
    public static printId(animal: IAnimal): void {
        // для успешного выполнения этого метода
        // не требуется доступ к данным о animal.age
        console.log(animal.id);
    }

 public static printAge(animal: IAnimal): void {
        // для успешного выполнения этого метода
        // не требуется доступ к данным о animal.id
        console.log(animal.age);
    }
}
```

Поэтому в подобных ситуациях настоятельно рекомендуется *“дробить”* типы интерфейсов на меньшие составляющие и затем ограничивать ими доступ к данным.

ts

```
interface IIdentifiable {}
interface ILiving {}

interface IAnimal extends IIdentifiable, ILiving {/*...*/}

class Animal implements IAnimal {/*...*/}
```

```
class AnimalUtil {
    public static printId(animal: IIdentifiable): void {
        // параметр animal ограничен типом IIdentifiable
        console.log(animal.id);
    }

    public static printAge(animal: ILiving): void {
        // параметр animal ограничен типом ILiving
        console.log(animal.age);
    }
}

class Bird extends Animal {}
class Raven extends Bird {/*...*/}

let raven: Raven = new Raven();

AnimalUtil.printId(raven);
AnimalUtil.printAge(raven);
```

[21.08] Расширение интерфейсом класса (extends class)

В случаях, когда требуется создать интерфейс для уже имеющегося класса, нет необходимости тратить силы на перечисление членов класса в интерфейсе. В *TypeScript* интерфейсу достаточно расширить тип класса.

Когда интерфейс расширяет класс, он наследует описание членов, но не их реализацию.

ts

```
class Animal {
    nickname: string;
    age: number;
}

interface IAnimal extends Animal {}

class Bird implements IAnimal {
    nickname:string;
    age: number;
}

let bird: IAnimal = new Bird();
```

Но с расширением класса интерфейсом существует один нюанс.

Интерфейс, полученный путем расширения типа класса, может быть реализован только самим этим классом или его потомками, поскольку помимо публичных (`public`) также наследует закрытые (`private`) и защищенные (`protected`) члены.

ts

```
class Animal {
    private uid: string;
    protected maxAge: number;
    public name: string;
}

interface IAnimal extends Animal {}

class Bird extends Animal implements IAnimal { // Ok
    // private uid: string = ''; // Error, private
    protected maxAge: number = 100; // Ok, protected
    public name: string = 'bird'; // Ok,  public
}

class Fish implements IAnimal { // Error
    public name: string = 'fish';
}

let bird: IAnimal = new Bird(); // Ok
let fish: IAnimal = new Fish(); // Error
```

[21.09] Описание класса (функции-конструктора)

Известный факт, что в *JavaScript*, а следовательно и в *TypeScript*, конструкция `class` — это лишь *“синтаксический сахар”* над старой доброй функцией-конструктором. Эта особенность позволяет описывать интерфейсы не только для экземпляров класса, но и для самих классов (функций-конструкторов). Проще говоря, с помощью интерфейса можно описать как конструктор, так и статические члены класса, с одной оговоркой — этот интерфейс можно использовать только в качестве типа. То есть класс не может указывать реализацию такого интерфейса с помощью ключевого слова `implements` сопряженную с экземпляром, а не самим классом.

Описание интерфейса для функции конструктора может потребоваться когда в качестве значения выступает сам класс.

Конструктор указывается с помощью ключевого слова `new` , затем открываются скобки, в которых (при наличии) указываются параметры, а в конце указывается тип возвращаемого значения.

ts

```ts
new(p1: type, p2: type): type;
```

Статические члены описываются так же, как и члены экземпляра.

ts

```ts
interface IAnimal {
    nickname: string;
}

class Animal implements IAnimal {
    nickname: string;

    constructor(nickname: string) {
        this.nickname = nickname;
    }
}

class Bird extends Animal {
    static DEFAULT_NAME: string = 'bird';

    static create(): IAnimal {
        return new Bird(Bird.DEFAULT_NAME);
```

```
    }
}

class Fish extends Animal {
    static DEFAULT_NAME: string = 'fish';

    static create(): IAnimal {
        return new Fish(Fish.DEFAULT_NAME);
    }
}

const bird: Bird = new Bird('bird');
const fish: Fish = new Fish('fish');

let a: IAnimal[] = [bird, fish]; // Ok, массив экземпляров классов
реализующих интерфейс IAnimal
let b: IAnimal[] = [Bird, Fish]; // Error, массив классов

interface IAnimalConstructor { // декларация интерфейса для класса
    create(): IAnimal; // static method
    new (nickname: string): IAnimal; // конструктор
}

let c: IAnimalConstructor[] = [Bird, Fish]; // Ok, массив классов
let d: IAnimal[] = c.map(item => item.create()); // Ok, массив
экземпляров классов реализующих интерфейс IAnimal
```

[21.10] Описание функционального выражения

Помимо экземпляров и самих классов, интерфейсы могут описывать функциональные выражения. Это очень удобно, когда функциональный тип имеет большую сигнатуру, которая делает код менее читабельным.

ts

```
// reduce(callbackFn: (previousValue: T, currentValue: T, currentIndex:
number, array: T[]) => T, initialValue?: T): T;
var callback: (previousValue: number, currentValue: number,
currentIndex: number, array: number[]) => number;
```

В большинстве подобных случаев можно прибегнуть к помощи вывода типов.

ts

```
// reduce(callbackFn: (previousValue: T, currentValue: T, currentIndex:
number, array: T[]) => T, initialValue?: T): T;

var callback: (previousValue: number, currentValue: number,
currentIndex: number, array: number[]) => number;
var callback = (previousValue: number, currentValue: number,
currentIndex: number, array: number[]) => previousValue + currentValue;

let numberAll: number[] = [5, 5, 10, 30];

let sum: number = numberAll.reduce(callback); // 50
```

Но в случае, если функциональное выражение является параметром функции, как например метод массива `reduce`, то решением может служить только явная декларация типа.

ts

```
class Collection<T> {
    reduce(callbackFn: (previousValue: T, currentValue: T, currentIndex:
number, array: T[]) => T, initialValue?: T): T {
        return null;
    }
}
```

Поэтому при необходимости указать тип явно, помимо рассмотренного в главе "Типы - Type Queries (запросы типа), Alias (псевдонимы типа)" механизма создания псевдонимов типа (`type`), можно описать функциональное выражение с помощью интерфейса.

Для этого необходимо в теле интерфейса описать сигнатуру функции без указания идентификатора.

ts

```
interface ISumAll {
    (...valueAll: number[]): number;
}

const sumAll: ISumAll = (...valueAll: number[]) =>
    valueAll.reduce((result, value) => result += value, 0);


let numberAll: number[] = [5, 5, 10, 30];

let sum: number = sumAll(...numberAll);
```

[21.11] Описание индексных членов в объектных типов

Индексные члены подробно будут рассматриваться в главе “Типы - Объектные типы с индексными членами (объектный тип с динамическими ключами)”, но не будет лишним и здесь коснутся этого механизма.

ts

```ts
interface Identifier {
    [BindingIdentifier: string]: Type;
    [BindingIdentifier: number]: Type;
}
```

[21.12] Инлайн интерфейсы (Inline Interface)

Помимо описания объекта в конструкции, объявляемой с помощью ключевого слова `interface`, тип объекта можно описать прямо в месте указания типа. Такой способ объявления типа неформально обозначается как *инлайн интерфейс* (*inline interface*). Всё ранее описанное для типов интерфейсов, объявленных с помощью ключевого слова `interface`, в полной мере верно и для их инлайн аналогов.

Различие между ними заключается в том, что второй обладает только телом и объявляется прямо в аннотации типа.

ts

```ts
let identifier: { p1: type, p2: type };
```

Интерфейс, объявленный с помощью ключевого слова `interface`, считается идентичным инлайн интерфейсу, если их описание совпадает. Но стоит обратить внимание, что это возможно благодаря структурной типизации, которая рассматривается в главе “Экскурс в типизацию - Совместимость типов на основе вида типизации”.

ts

```ts
interface IAnimal {
    nickname: string;
}

class Bird implements IAnimal {
    nickname: string;
}
class Fish implements IAnimal {
    nickname: string;
}

let bird: IAnimal = new Bird(); // Ok
let fish: { nickname: string } = new Fish(); // Ok
```

Как было сказано ранее, инлайн интерфейс можно объявлять в тех местах, в которых допускается указание типа. Тем не менее, реализовывать (`implements`) и расширять (`extends`) инлайн интерфейс нельзя.

ts

```ts
interface IT1 {}
interface IT2 {}

interface IT3 extends { f1: IT1, f2: IT2 } { // Error

}

class T4 implements { f1: T1, f2: T2 } { // Error

}
```

Хотя последнее утверждение и не совсем верно. В дальнейшем будет рассказано о такой замечательной конструкции, как обобщения (глава “Типы - Обобщения (Generics)”), в которых, как раз таки возможно расширять (`extends`) инлайн интерфейсы.

[21.13] Слияние интерфейсов

В случае, если в одной области видимости объявлено несколько одноимённых интерфейсов, то они будут объединены в один.

ts

```
// так видят разработчики
interface IAnimal {
    name: string;
}

interface IAnimal {
    age: number;
}

//так видит компилятор
/**
interface IAnimal {
    name: string;
    age: number;
}
*/

// разработчики получают то, что видит компилятор
let animal: IAnimal;
animal.name = 'animal'; // Ok
animal.age = 0; // Ok
```

При попытке переопределить тип поля, возникнет ошибка.

ts

```
interface IAnimal {
    name: string;
    age: number;
}

interface IAnimal {
    name: string; // Ok
    age: string; // Error
}
```

Если в нескольких одноимённых интерфейсах будут описаны одноимённые методы с разными сигнатурами, то они будут расценены, как описание перегрузки. К тому же, интерфейсы, которые описывают множество одноимённых методов, сохраняют свой внутренний порядок.

ts

```
interface IBird {}
interface IFish {}
interface IInsect {}
interface IReptile {}

// до компиляции
interface IAnimalFactory {
    getAnimalByID(id: number): IBird;
```

```
}

interface IAnimalFactory {
    getAnimalByID(id: string): IFish;
}

interface IAnimalFactory {
    getAnimalByID(id: boolean): IInsect;
    getAnimalByID(id: object): IReptile;
}

/** при компиляции
interface IAnimalFactory {
    getAnimalByID(id: string): IInsect;
    getAnimalByID(id: string): IReptile;
    getAnimalByID(id: string): IFish;
    getAnimalByID(id: string): IBird;
}
*/
let animal: IAnimalFactory;
let v1 = animal.getAnimalByID(0); // Ok -> v1: IBird
let v2 = animal.getAnimalByID('5'); // Ok -> v2: IFish
let v3 = animal.getAnimalByID(true); // Ok -> v3: IInsect
let v4 = animal.getAnimalByID({}); // Ok -> v4: IReptile
```

Исключением из этого правила являются сигнатуры, которые имеют в своем описании литеральные строковые типы данных (`literal String Types`). Дело в том, что сигнатуры содержащие в своем описании литеральные строковые типы, всегда размещаются перед сигнатурами, у которых нет в описании литеральных строковых типов.

ts

```
interface IBird {}
interface IFish {}
interface IInsect {}
interface IReptile {}

// до компиляции
interface IAnimalFactory {
    getAnimalByID(id: string): IBird;
}

interface IAnimalFactory {
    getAnimalByID(id: 'fish'): IFish;
}

interface IAnimalFactory {
    getAnimalByID(id: 'insect'): IInsect;
    getAnimalByID(id: number): IReptile;
}
```

```
/** при компиляции
interface IAnimalFactory {
    getAnimalByID(id: 'fish'): IFish;
    getAnimalByID(id: 'insect'): IInsect;
    getAnimalByID(id: number): IReptile;
    getAnimalByID(id: string): IBird;
}
*/
```

Глава 22

Объектные типы с индексными членами (объектный тип с динамическими ключами)

Впервые реализуя динамические ключи в статически типизированном *TypeScript*, могут возникнуть трудности, которые вместе с сопряженными тонкостями, будут подробно рассмотрены в этой главе.

[22.0] Индексные члены (определение динамических ключей)

Статический анализ кода всеми силами стремится взять под контроль синтаксические конструкции, тем самым переложить работу, связанную с выявлением ошибок, на себя, оставляя разработчику больше времени на более важные задачи. И несмотря на то, что динамические операции являются причиной “головной боли” компилятора, потребность в них при разработке программ все-таки существует. Одной из таких операций является определение в объектах *индексных членов* (динамических ключей).

Индексная сигнатура (*index signature*) состоит из двух частей. В первой части расположен имеющий собственную аннотацию типа *идентификатор привязки* (*binding identifier*) заключенный в квадратные скобки `[]`. Во второй части расположена *аннотация типа* (*type annotation*) представляющего значение ассоциируемое с динамическим ключом.

ts

```
{ [identifier: Type]: Type }
```

Идентификатору привязки можно дать любое имя, которое должно быть ассоциировано только с типами `string`, `number`, `symbol` или `literal template string`, а в качестве типа указанного справа от двоеточия, может быть указан любой тип.

ts

```ts
// именование ключа - идентификатор ключа может быть любым
interface Identifier {
    [identifier: string]: string; // идентификатор - identifier
}
// или
interface Identifier {
    [someKey: string]: string; // идентификатор - someKey
}

// допустимые типы - string, number, symbol или litelral template string
interface Identifier {
    [key: string]: string; // будет соответствовать o[`key`]
}
interface Identifier {
    [key: number]: string; // будет соответствовать o[5]
}
interface Identifier {
    [key: symbol]: string; // будет соответствовать o[Symbol(`key`)]
}
interface Identifier {
    [key: `data-${string}`]: string; // будет соответствовать o[`data-
*`]
}
```

В одном объектном типе одновременно могут быть объявлены индексные сигнатуры, чьи идентификаторы привязки принадлежат к типу `string`, `number`, `symbol` или `literal template string`. Но с одной оговоркой. Их типы, указанные в аннотации типов, должны быть совместимы (совместимость типов подробно рассматривается в главах “Типизация - Совместимость объектов” и “Типизация - Совместимость функций”).

ts

```ts
interface A {
    [key: string]: string;
    [key: number]: string;
    [key: symbol]: string;
    [key: `data-${string}`]: string;
}

let a: A = {
    validKeyDeclareStatic: 'value', // Ok, значение принадлежит к string
```

```
    invalidKeyDeclareStatic: 0 // Error, значение должно быть
совместимым с типом string
};

a.validKeyDefineDynamicKey = 'value'; // Ok
a.invalidKeyDefineDynamicKey = 0; // Error, значение должно быть
совместимым с типом string
a[0] = 'value'; // Ok

interface B {
    [identifier: string]: string; // Ok
    [identifier: string]: string; // Error, дубликат
}

interface C {
    [identifier: string]: string; // Ok
    [identifier: number]: number; // Error, должен принадлежать к типу
string
}

class SuperClass { // суперкласс
    a: number;
}

class SubClass extends SuperClass { // подкласс
    b: number;
}

interface D {
    [identifier: string]: SuperClass; // Ok
    [identifier: number]: SubClass; // Ok, SubClass совместим с
SuperClass
}

let d: D = {};
d.dynamicKey = new SubClass(); // Ok
d[0] = new SubClass(); // Ok

interface E {
    [identifier: string]: SubClass; // Ok
    [identifier: number]: SuperClass; // Error, SuperClass несовместим
с SubClass
}
```

Множественное определение типов, к которым могут принадлежать ключи индексной сигнатуры, можно записать также с помощью типа объединение.

ts

```
interface A {
    [key: string | number | symbol | `data-${string}`]: string; // это
тоже самое, что и...
```

```
}
interface A { // ...это
    [key: string]: string;
    [key: number]: string;
    [key: symbol]: string;
    [key: `data-${string}`]: string;
}
```

Так как классы принадлежат к объектным типам, их тела также могут определять индексные сигнатуры, в том числе и уровня самого класса, то есть - статические индексные сигнатуры (`static`). Правила, относящиеся к индексным сигнатурам, которые были и будут рассмотрены в этой главе, в полной мере справедливы и для классов.

ts

```ts
class Identifier {
  static [key: string]: string; // статическая индексная сигнатура


  [key: string]: string;
  [key: number]: string;

  [0]: 'value';
  [1]: 5; // Error, все члены должны принадлежать к совместимым со
string типам

  public a: string = 'value'; // Ok, поле name с типом string
  public b: number = 0; // Error, все члены должны принадлежать к
совместимым со string типам

  public c(): void {} // Error, метод тоже член и на него
распространяются те же правила
}

let identifier: Identifier = new Identifier();

/**индексная сигнатура экземпляра класса */
identifier.validDynamicKey = 'value'; // Ok
identifier.invalidDynamicKey = 0; // Error

identifier[2] = 'value'; // Ok
identifier[3] = 0; // Error

/**индексная сигнатура класса */
Identifier.validDynamicKey = 'value'; // Ok
Identifier.invalidDynamicKey = 0; // Error

Identifier[2] = 'value'; // Ok
Identifier[3] = 0; // Error
```

Кроме того, классы накладывают ограничение не позволяющее использовать модификаторы доступа (`private` , `protected` , `public`). При попытке указать данные модификаторы для индексной сигнатуры возникнет ошибка.

ts

```ts
class A {
    public [key: string]: string; // Error
}
```

Но, относительно модификаторов, есть несколько нюансов, связанных с модификатором `readonly` , который подробно рассматривается в главе [“Классы - Модификатор readonly”](). Чтобы ничего не ускользнуло от понимания, начнем по порядку.

При указании модификатора `readonly` искажается смысл использования индексной сигнатуры, так как это дает ровно противоположный эффект. Вместо объекта с возможностью определения динамических членов получается объект, позволяющий лишь объявлять динамические ключи, которым нельзя ничего присвоить. То есть, объект полностью закрыт для изменения.

В случае с интерфейсом:

ts

```ts
interface IIdentifier {
    readonly [key: string]: string; // Ok, модификатор readonly
}

let instanceObject: IIdentifier = {};

instanceObject.a; // Ok, можно объявить
instanceObject.a = 'value'; // Error, но нельзя присвоить значение
```

В случае с классом:

ts

```ts
class Identifier {
    readonly [key: string]: string;
}

let instanceClass = new Identifier();
instanceClass.a; // Ok, можно объявить
instanceClass.a = 'value'; // Error, но нельзя присвоить значение
```

Второй нюанс заключается в том, что если в объекте или классе определена индексная сигнатура, становится невозможным объявить в их теле или добавить через точечную и скобочную нотацию ключи, ассоциированные с несовместимым типом данных.

Простыми словами — тело объекта или класса, имеющего определение индексной сигнатуры, не может иметь определения членов других типов.

В случае с интерфейсом:

ts

```ts
interface IIdentifier {
    [key: string]: string;

    a: string; // Ok, [в момент декларации]
    b: number; // Error, [в момент декларации] допускается объявление
идентификаторов принадлежащих только к типу string
}

let instanceObject: IIdentifier = {
    c: '', // Ok, [в момент объявления]
    d: 0   // Error, [в момент объявления] допускается объявление
идентификаторов принадлежащих только типу string
};

instanceObject.e = ''; // Ok, [после объявления]
instanceObject.f = 0;  // Error, [после объявления] допускается
объявление идентификаторов принадлежащих только типу string
```

В случае с классом:

ts

```ts
class Identifier {
    [key: string]: string;

    a: string; // Ok, [в момент объявления]
    b: number; // Error, [в момент объявления] допускается объявление
идентификаторов принадлежащих только типу string
}

let instanceClass = new Identifier();
instanceClass.c = ''; // Ok, [после объявления]
instanceClass.d = 0;  // Error, [после объявления] допускается
объявление идентификаторов принадлежащих только типу string
```

Но, в случае с модификатором `readonly`, поведение отличается. Несмотря на то, что указывать идентификаторы членов, принадлежащие к несовместимым типам, по-прежнему нельзя, допускается их декларация и объявление.

В случае с интерфейсом:

ts

```
interface IIdentifier {
    readonly [key: string]: string; // Ok

    a: string; // Ok, декларация
}

let instanceObject: IIdentifier = {
    a: '', // Ok, объявление
    b: ''  // Ok, объявление
};

instanceObject.c = 'value'; // Error, ассоциировать ключ со значением
после создания объекта по-прежнему нельзя
```

В случае с классом:

ts

```
class Identifier {
    readonly [key: string]: string;

    a: string = 'value'; // Ok, декларация и объявление
}

let instanceClass = new Identifier();
instanceClass.b = 'value'; // Error, ассоциировать ключ со значением
после создания объекта по-прежнему нельзя
```

К тому же объекты и классы имеющие определение индексной сигнатуры не могут содержать определения методов.

ts

```
interface IIdentifier {
    readonly [key: string]: string;

    method(): void; // Error -> TS2411: Property 'method' of type '()
=> void' is not assignable to string index type 'string'
}

class Identifier {
    readonly [key: string]: string;

    method(): void {} // Error -> TS2411: Property 'method' of type '()
=> void' is not assignable to string index type 'string'.
}
```

Третий нюанс проистекает от предыдущего и заключается в том, что значения, ассоциированные с идентификаторами, которые были задекларированы в типе, можно перезаписать после инициализации объекта.

В случае с интерфейсом:

ts

```ts
interface IIdentifier {
    readonly [key: string]: string; // Ok

    a: string; // Ok, декларация
}

let instanceObject: IIdentifier = {
    a: 'value', // Ok, реализация
    b: 'value'  // Ok, объявление
};

instanceObject.a = 'new value'; // Ok, можно перезаписать значение
instanceObject.b = 'new value'; // Error, нельзя перезаписать значение
```

В случае с классом:

ts

```ts
class Identifier {
    readonly [key: string]: string;

    a: string = 'value'; // Ok, декларация и объявление
}

let instanceClass = new Identifier();
instanceClass.a = 'new value'; // Ok, можно перезаписать значение
```

Если учесть, что модификатор `readonly` вообще не стоит применять к индексной сигнатуре, то все эти нюансы вообще можно выкинуть из головы. Но, так как цель этой книги защитить читателя от как можно большего количества подводных камней, я решил не опускать данный факт, ведь знание — лучшая защита.

Кроме того, не будет лишним знать наперед, что если идентификатор привязки принадлежит к типу `string`, то в качестве ключа может быть использовано значение, принадлежащее к типам `string`, `number`, `symbol`, `Number Enum` и `String Enum`.

ts

```ts
interface StringDynamicKey {
    [key: string]: string;
}

enum NumberEnum {
    Prop = 0
}
```

```
enum StringEnum {
    Prop = 'prop'
}

let example: StringDynamicKey = {
    property: '',            // Ok String key
    '': '',                  // Ok String key
    1: '',                   // Ok Number key
    [Symbol.for('key')]: '', // Ok Symbol key
    [NumberEnum.Prop]: '',   // Ok Number Enum key
    [StringEnum.Prop]: '',   // Ok String Enum key
};
```

В случае, когда идентификатор привязки принадлежит к типу `number`, то значение, используемое в качестве ключа, может принадлежать к таким типам, как `number`, `symbol`, `Number Enum` и `String Enum`.

ts

```
interface NumberDynamicKey {
    [key: number]: string;
}

enum NumberEnum {
    Prop = 0
}

enum StringEnum {
    Prop = 'prop'
}

let example: NumberDynamicKey = {
    property: '',               // Error String key
    '': '',                     // Error String key
    1: '',                      // Ok Number key
    [Symbol.for('key')]: '',    // Ok Symbol key
    [NumberEnum.Prop]: '',      // Ok Number Enum key
    [StringEnum.Prop]: '',      // Ok String Enum key
};
```

Вывод типов, в некоторых случаях, выводит тип, принадлежащий к объектному типу с индексной сигнатурой. Напомню, что в *JavaScript*, помимо привычного способа при объявлении идентификаторов в объектных типах, можно использовать строковые литералы и выражения заключённые в квадратные скобки `[]`.

ts

```
let computedIdentifier = 'e';

let v = {
    a: '', // объявление идентификатора привычным способом,
```

```
    ['b']: '', // объявление идентификатора с помощью строкового
литерала.
    ['c' + 'd']: '', // объявление идентификатора с помощью выражения
со строковыми литералами
    [computedIdentifier]: '' // объявление идентификатора при помощи
вычисляемого идентификатора
};
```

В первых двух случаях, вывод типов выводит ожидаемый тип, а в оставшихся тип с индексной сигнатурой.

ts

```
// let v1: { a: string; }
let v1 = {
    a: 'value'  // Ok, привычный идентификатор
};

v1.b = 'value'; // Error, в типе { a: string } не задекларирован
идентификатор b
```

ts

```
// let v2: { ['a']: string; }
let v2 = {
    ['a']: 'value'  // Ok, строковый литерал
};

v2.b = 'value'; // Error, в типе { ['a']: string } не задекларирован
идентификатор b
```

ts

```
let computedIdentifier: string = 'a';

// let v3: { [x: string]: string }; - вывод типов выводит как тип с
индексной сигнатурой...
let v3 = {
    [computedIdentifier]: 'value' // вычисляемое свойство
};

v3.b = 'value'; // ... а это, в свою очередь, позволяет добавлять новое
значение
```

ts

```
// let v4: { [x: string]: string }; - вывод типов выводит как тип с
индексной сигнатурой...
let v4 = {
    ['a' + 'b']: 'value' // выражение со строковыми литералами
};
```

```
v4.b = 'value'; // ... а это, в свою очередь, позволяет добавлять новое
значение
```

[22.1] Строгая проверка при обращении к динамическим ключам

Поскольку механизм, позволяющий определение *индексной сигнатуры*, не способен отслеживать идентификаторы (имена) полей определенных динамически, такой подход не считается типобезопасным.

ts

```ts
type T = {
    [key: string]: number | string;
}

function f(p: T) {
    /**
     * Обращение к несуществующим полям
     */
    p.bad.toString(); // Ok -> Ошибка времени исполнения
    p[Math.random()].toString(); // Ok -> Ошибка времени исполнения
}
```

Данная проблема решается с помощью флага `--noUncheckedIndexedAccess` активирующего строгую проверку при обращении к динамическим членам объектных типов. Флаг `--noUncheckedIndexedAccess` ожидает в качестве значения `true` либо `false`. Активация механизма позволяет обращаться к динамическим членам только после подтверждения их наличия в объекте, а также совместно с такими операторами, как оператор опциональной последовательности `?.` и опциональный оператор `!.`.

json

```json
// @filename: tsconfig.json

{
    "compilerOptions": {
        "noUncheckedIndexedAccess": true
    }
}
```

ts

```ts
type T = {
  [key: string]: number | string;
}


function f(p: T) {
  /**
   * Обращение к несуществующим полям
   */
  p.bad.toString(); // Error -> TS2532: Object is possibly 'undefined'.
  p[Math.random()].toString(); // Error -> TS2532: Object is possibly
'undefined'.


  // Проверка наличия поля bad
  if ("bad" in p) {
      p.bad?.toString(); // Ok
  }

  // Использование опционального оператора
  p[Math.random()]!.toString(); // Ok -> ошибка во время выполнения

  p[Math.random()]?.toString();  // Ok -> Ошибка не возникнет
}
```

Не будет лишним упомянуть, что влияние данного механизма распространяется также и на массивы. В случае с массивом не получится избежать аналогичной ошибки при попытке обращения к его элементам при помощи индексной сигнатуры.

ts

```ts
function f(array: string[]) {
    for (let i = 0; i < array.length; i++) {
        array[i].toString(); // Error -> TS2532: Object is possibly
'undefined'.
    }
}
```

[22.2] Запрет обращения к динамическим ключам через точечную нотацию

Поскольку к динамическим ключам можно обращаться, как через точечную, так и скобочную нотацию, то может возникнуть потребность в разделении такого поведения. Для подобных случаев, в *TypeScript* существует флаг `--noPropertyAccessFromIndexSignature`, установление которого в значение `true`, запрещает обращение к динамическим членам с помощью точечной нотации.

json

```
// @filename: tsconfig.json

{
    "compilerOptions": {
        "noPropertyAccessFromIndexSignature": "true"
    }
}
```

ts

```
type Settings = {
    env?: string[]; // определение необязательного предопределенного
поля

    [key: string]: any; // определение динамических полей
}


function configurate(settings: Settings){
    //---------------------------
    // динамическое поле
    if(settings.envs){ // Ошибка при активном флаге и Ok при не активном

    }
    if(settings['envs']){ // Ok при любом значении флага

    }

    //---------------------------
    // предопределенное поле
    if(settings.env){ // Ok [1]

    }
```

```
    if(settings['env']){ // Ok при любом значении флага

    }
}
```

[22.3] Тонкости совместимости индексной сигнатурой с необязательными полями

Объектные типы определяющие строковую индексную сигнатуру считаются совместимыми с объектными типами имеющими необязательные поля.

ts

```ts
/**
 * [0] поля определены как необязательные!
 */
type Magic = {
    fire?: string[];
    water?: string[];
}

declare const HERO_CONFIG: Magic;

const hero: {[key: string]: string[]} = HERO_CONFIG; // Ok
/**Ok*/
```

Но существует два неочевидных момента. При попытке инициализировать необязательные поля значением `undefined` возникнет ошибка.

ts

```ts
/**
 * [0] поля определены как необязательные!
 */
type Magic = {
    fire?: string[];
    water?: string[];
}

declare const HERO_CONFIG: Magic;

/**
 * [1] Error ->
```

```
 * Type 'undefined' is not assignable to type 'string[]'.
 */
const hero: {[key: string]: string[]} = HERO_CONFIG;
hero['fire'] = ['fireball']; // Ok
hero['water'] = undefined; // Error [1]
```

Кроме этого, ошибки не удастся избежать тогда, когда объектный тип, вместо опционального оператора, явно указывает принадлежность типа членов к типу `undefined` .

ts

```
/**
 * [0] поля определены как обязательные!
 */
type Magic = {
    fire: string[] | undefined; // [0]
    water: string[] | undefined; // [0]
}

declare const HERO_CONFIG: Magic;

/**
 * [1] Error ->
 * Type 'Magic' is not assignable to type '{ [key: string]:
string[]; }'.
 */
const hero: {[key: string]: string[]} = HERO_CONFIG;
/**[1]*/
```

И, кроме того, данные правила применимы исключительно к строковой индексной сигнатуре.

ts

```
/**
 * [0] ключи полей определены как индексы!
 */
type Port = {
    80?: string; // [0]
    88?: string; // [0]
}

declare const SERVER_PORT: Port;

/**
 * [2] Ключ индексной сигнатуры принадлежит к типу number.
 *
 * [1] Error ->
 * Type 'Port' is not assignable to type '{ [key: number]: string[]; }'.
 */
```

```
const port: {[key: number]: string[]} = SERVER_PORT;
   /**[1] */     /**[2] */
```

Глава 23

Модификаторы доступа (Access Modifiers)

Поскольку сокрытие является одним из основных принципов структурного и объектно-ориентированного проектирования, при создании программ принято скрывать как можно больше информации об объектах. Это необходимо не только для предотвращения случайного изменения, но и для повышения слабого зацепления (*low coupling*). Для этих целей *TypeScript* реализует механизм, называемый *модификаторы доступа*.

Модификаторы доступа — это ключевые слова, с помощью которых осуществляется управление сокрытием данных в классе. Простыми словами, модификаторы доступа задают уровень доступности членам класса ограничивая область их видимости.

В *TypeScript* существует три модификатора доступа, указывающихся с помощью ключевых слов `public`, `protected` и `private`. Влияние модификаторов доступа ограничивается *TypeScript* и после компиляции от них не остается ни следа. В скомпилированном коде нет никакой разницы между членами, к которым были применены те или иные модификаторы доступа.

С помощью модификаторов доступа можно указать уровень доступа для полей, аксессоров и методов, в том числе статических. Кроме того, в *TypeScript* при помощи модификаторов доступа можно ограничивать доступ к конструктору класса.

ts

```ts
class Animal {
    private static DEFAULT_NICKNAME: string = 'animal';

    protected static inputInfoDecor(text: string): string {
        return `[object ${text}]`;
    }

    private _nickname: string = Animal.DEFAULT_NICKNAME;
```

```
    public get nickname(): string {
        return this._nickname;
    }

    public set nickname(value: string){
        this._nickname = value;
    }

    public constructor() {}

    public toString(): string {
        return Animal.inputInfoDecor('Animal');
    }
}
```

[23.0] Модификатор доступа public (публичный)

Члены, помеченные ключевым словом `public`, доступны в определяющих их классах, их потомках, а также к ним можно обращаться через экземпляр или, в случае статических членов, через ссылку на класс.

Если при разработке приложения особое внимание уделяется архитектуре, то, как правило, модификатором доступа `public` помечаются только члены описанные в интерфейсе (глава “Типы - Interface”).

ts

```
class Animal {
    public nickname: string;

    constructor() {
        this.nickname = 'animal';
    }
}

class Bird extends Animal {
  constructor() {
      super();
      super.nickname = 'bird';
  }
}

let animal: Animal = new Animal();
animal.nickname = 'newAnimal';
```

```
let bird: Bird = new Bird();
bird.nickname = 'newBird';
```

Члены, у которых отсутствует указание какого-либо модификатора доступа, воспринимаются компилятором как `public`:

ts

```
class Animal {
 nickname: string; // эквивалентно public nickname: string
}
```

[23.1] Модификатор доступа private (закрытый или скрытый)

Модификатор доступа `private` является полной противоположностью модификатору доступа `public`. Члены, помеченные с помощью ключевого слова `private`, доступны только контексту класса, в котором они определены.

Чем меньше окружающему миру известно о внутреннем устройстве классов, тем меньше оно взаимодействует с ним. Взаимодействия программы с объектом называют *сопряжением* (coupling), уровень которого варьируется от сильного до слабого. *Слабое сопряжение* (*loose coupling*) является признаком качественной архитектуры программы. Этот факт подталкивает скрывать как можно больше информации при проектировании программы. В языках программирования, в которых существуют модификаторы доступа, скрывать различные конструкции принято с помощью модификатора `private`.

ts

```
class Animal {
    private metainfo: string;

    constructor() {
        this.metainfo = '...';
    }
}

class Bird extends Animal {
    constructor() {
        super();
        super.metainfo = 'bird'; // Error
```

```
    }
}

let animal: Animal = new Animal();
animal.metainfo = 'newAnimal'; // Error

let bird: Bird = new Bird();
bird.metainfo = 'newBird'; // Error
```

[23.2] Модификатор доступа protected (защищенный)

Члены, которым установлен модификатор доступа `protected`, доступны только контексту класса, в котором они определенны, а также всем его потомкам. Попытка обратится к членам, помеченным как `protected` снаружи, приведет к возникновению ошибки.

Как правило, при современной разработке упор делается только на два вида модификаторов доступа - `public` и `private`. Но, в редких случаях, по сценарию может возникнуть необходимость обращаться к членам своего суперкласса в подклассах, которые при этом, должны быть скрыты от окружающего мира. В таких случаях члены помечают как `protected`.

ts

```
class Animal {
    protected isUpdate: boolean;

    constructor() {
        this.isUpdate = false;
    }
}

class Bird extends Animal {
    constructor() {
        super();
        super.isUpdate = false;
    }
}

let animal: Animal = new Animal();
animal.isUpdate = true; // Error
```

```
let bird: Bird = new Bird();
bird.isUpdate = true; // Error
```

[23.3] Модификаторы доступа и конструкторы класса

В *TypeScript* существует возможность управлять доступом конструкторов. Конструктор, который не отмечен ни одним модификатором доступа эквивалентен конструктору, помеченному как `public`.

ts

```
class A {
    public constructor() {} // модификатор public установлен явно
}

class B {
    constructor() {} // модификатор public установлен не явно
}
```

Если класс состоит только из статических свойств и методов, как например класс `Math`, то его конструктор более разумно пометить как приватный и, тем самым, запретить создание его экземпляров.

ts

```
class AnimalUtil {
    private static MS_TO_DAY: number = 1000 * 60 * 60 * 24;

    public static ageFromMsToDayFormat(time: number): number {
        return Math.ceil(time / AnimalUtil.MS_TO_DAY);
    }

    private constructor() {};
}

class Animal {
    private _timeOfBirth: number = Date.now();

    public get age(): number {
        return this._timeOfBirth - Date.now();
    }
```

```
    public constructor() {}
}

let animal: Animal = new Animal();
let age: number = AnimalUtil.ageFromMsToDayFormat(animal.age);

let util = new AnimalUtil(); // Ошибка при компиляции, нельзя создать
экземпляр
```

Кроме того, класс, у которого конструктор объявлен с модификатором доступа `private`, нельзя расширять (`extends`).

ts

```
class AnimalUtil {
    // ...

    private constructor() {};
}

class AnimalStringUtil extends AnimalUtil {} // Ошибка
```

Класс, в котором конструктор объявлен с модификатором доступа `protected`, как и в случае с модификатором доступа `private`, также не позволит создать свой экземпляр, но открыт для расширения (`extends`).

ts

```
class Animal {
    protected constructor() {}
}

// можно наследоваться от класса с защищенным конструктором
class Bird extends Animal {
    constructor() {
        super();
    }
}

let animal: Animal = new Animal(); // Error
let bird: Bird = new Bird(); // Ok
```

Тем не менее есть один нюанс. Не получится создать экземпляр подкласса, если он не переопределил конструктор суперкласса.

ts

```
class Animal {
    protected constructor() {}
}
```

```
// потомок переопределяет конструктор предка
class Bird extends Animal {
    constructor() {
        super();
    }
}

// потомок не переопределяет конструктор предка
class Fish extends Animal {}

let bird: Bird = new Bird(); // Ok
let fish: Fish = new Fish(); // Error
```

[23.4] Быстрое объявление полей

В разработке очень часто используются классы, состоящие только из открытых полей, ассоциированных со значением, переданным в качестве аргументов конструктора.

Помимо обычного способа объявления полей в теле класса с последующей инициализацией в конструкторе, существует сокращенный вариант с объявлением непосредственно в конструкторе за счет добавления его параметрам модификаторов.

Обычный способ:

ts

```
class BirdEntity {
    public name: string;
    public age: number;
    public isAlive: boolean;

    constructor(name: string, age: number, isAlive: boolean){
        this.name = name;
        this.age = age;
        this.isAlive = isAlive;
    }
}
```

Сокращенный способ инициализации полей класса:

ts

```
class FishEntity {
    constructor(
```

```
        public name: string,
        public age: number,
        public isAlive: boolean
    ) {}
}
```

В сокращенном варианте поля не указываются в теле класса, а указываются непосредственно в конструкторе. При этом обязательным условием является указание одного из существующих модификаторов доступа (либо модификатора `readonly`, о котором речь пойдет в главе “Классы - Модификатор readonly”), после чего параметры начинают расцениваться компилятором в качестве полей класса.

ts

```ts
class FishEntity {
    constructor(
        public name: string, // поле
        public age: number, // поле
        isAlive: boolean // параметр
    ) {}
}
```

При этом не существует строго заданного порядка, в котором должны объявляться параметры и сокращенное объявление полей.

ts

```ts
class FishEntity {
    constructor(
        name: string, // параметр
        public age: number, // поле
        isAlive: boolean // параметр
    ) {}
}
```

Полям, объявленным в конструкторе так же, как и полям, объявленным в теле класса, можно устанавливать различные модификаторы доступа.

ts

```ts
class FishEntity {
    constructor(
        public name: string,
        protected age: number,
        private isAlive: boolean
    ) {}
}
```

Обращение к полям, объявленным прямо в конструкторе, ничем не отличается от обращения к полям, объявленным в теле класса.

ts

```ts
class Animal {
    constructor(
        public name: string,
        protected age: number,
        private isAlive: boolean
    ) {}

    public dead(): void {
        this.isAlive = false;
    }
}

class Bird extends Animal {
    public toString(): string {
        return `[bird ${super.name}]`;
    }
}
```

Глава 24

Закрытые поля определенные спецификацией ECMAScript

Помимо сокрытия полей класса от внешней среды с помощью модификатора доступа `private`, присущего исключительно *TypeScript*, существует возможность прибегнуть к механизму, предусмотренному спецификацией *ECMAScript*.

[24.0] Нативный закрытый (private) модификатор доступа

Для того, что бы поля класса оставались закрытыми после компиляции в *JavaScript*, необходимо прибегнуть к модификатору, определенному спецификацией *ECMAScript* `#` который указывается в качестве префикса их идентификаторов (имен) `#identifier`.

Доступ к защищенному полю класса ограничивается областью видимости класса в котором оно объявлено, а при обращении к нему необходимо также указывать символ решетки.

ts

```ts
class Animal {
    #isLife: boolean = true; // защищенное поле класса

    get isLife() {
        return this.#isLife;
```

```
    }
}

let animal = new Animal();
console.log(animal.isLife); // обращение к аксессору, а не защищенному
полю
```

Поскольку доступ ограничивается областью видимости класса, потомки не могут обращаться к защищенным полям своих предков.

ts

```
class Animal {
    #isLife: boolean = true; // защищенное поле класса
}

class Bird extends Animal {
    constructor() {
        super();
        this.#isLife; // Error, TS18013: Property '#isLife' is not
accessible outside class 'Animal' because it has a private identifier.
    }
}
```

Данный модификатор может быть применён не только ко всем членам экземпляра класса предусмотренного спецификацией *ECMAScript*, но и к членам самого класса (`static`).

ts

```
class T {
    /** члены класса */

    static #CLASS_FIELD = "";
    static get #classProp(){
        return T.#CLASS_FIELD;
    }
    static set #classProp(value: string){
    }
    static #classMethod(){

    }


    /** члены экземпляра класса */

    #instanceField = "";

    get #instanceProp(){
        return this.#instanceField;
    }
```

```
    set #instanceProp(value: string){
    }

    #instanceMethod(){

    }
}
```

Но стоит сразу сказать, что модификация статических полей, при активном флаге возможна только при активации флаге `--useDefineForClassFields` .

ts

```ts
class T {
  /** члены класса */
  /**
   * [*] Ok, если useDefineForClassFields = true
   * [*] Error, если useDefineForClassFields = false
   */

  static #CLASS_FIELD = ""; // [*]
}
```

В отличие от модификатора доступа `private` , этот механизм не может быть применён к методам и аксессорам класса, но, так как за его появлением стоит спецификация *ECMAScript*, он продолжает действовать в скомпилированной программе. Именно поэтому, в отличие от сценария с модификатором доступа `private` , *потомки* могут без страха нарушить ожидаемый ход выполнения программы и объявлять защищенные поля, чьи идентификаторы идентичны объявлениям в их *супер-классах*.

Сценарий с модификатором доступа private:

ts

```ts
class Animal {
    private _isLife: boolean = true;
}

/**
 * Error!
 *
 * TS2415: Class 'Bird' incorrectly extends base class 'Animal'.
 * Types have separate declarations of a private property '_isLife'
 */
class Bird extends Animal {
    private _isLife: boolean = false;
}
```

Сценарий с защищенными полями предусмотренными спецификацией *ECMAScript*:

ts

```ts
class Animal {
    #isLife: boolean = true;
}

/**
 * Ok!
 */
class Bird extends Animal {
    #isLife: boolean = false;
}
```

И в заключение, стоит упомянуть, что существует несколько нюансов — один из них заключается в том, что закрытые поля нельзя объявлять непосредственно в конструкторе.

ts

```ts
class Animal {
    constructor(#isLife = true) {} // Error, TS18009: Private
identifiers cannot be used as parameters.
}
```

Другой нюанс связан с тем, что код, содержащий закрытые поля класса, может быть скомпилирован исключительно в версии `es6` и выше.

Глава 25

Абстрактные классы (abstract classes)

Если у всех начинающих разработчиков при размышлениях об интерфейсах возникают вопросы *"когда и зачем их использовать"*, то при размышлении об абстрактных классах к ним добавляются *"чем они отличаются от интерфейсов и когда та или иная конструкция предпочтительней"*. Ответы на эти вопросы вы найдете в данной главе, но для начала стоит рассмотреть общие характеристики.

[25.0] Общие характеристики

В *TypeScript* объявление абстрактного класса отличается от объявления обычного только добавлением ключевого слова `abstract` перед ключевым словом `class` .

ts

```ts
abstract class Identifier {}
```

Абстрактные классы также, как и обычные классы, могут расширять другие обычные и абстрактные классы и реализовывать интерфейсы.

ts

```ts
interface IInterface {}
```

```
class StandardClass {}

 // абстрактный класс расширяет обычный класс и реализует интерфейс
abstract class SuperAbstractClass extends StandardClass implements
IInterface {}

// абстрактный класс расширяет другой абстрактный класс
abstract class SubAbstractClass extends SuperAbstractClass {}
```

Несмотря на то, что абстрактный класс — все же класс, главное его отличие от обычного класса заключается в отсутствии возможности создания его экземпляров. Другими словами, нельзя создать экземпляр абстрактного класса.

ts

```
abstract class SuperAbstractClass {}
class SubStandardClass extends SuperAbstractClass {}

let v0: SuperAbstractClass = new SuperAbstractClass(); // Error, нельзя
создавать экземпляры абстрактного класса
let v1: SuperAbstractClass = new SubStandardClass(); // Ok
let v2: SubStandardClass = new SubStandardClass(); // Ok
```

Абстрактные классы могут содержать абстрактные члены, принадлежность к которым указывается с помощью ключевого слова `abstract`. Ключевое слово `abstract` можно применить к полям, свойствам (аксессоры) и методам абстрактного класса. При этом свойства и методы не должны иметь реализацию. В отличие от них, полям, помеченным как абстрактные, может быть присвоено значение по умолчанию.

ts

```
abstract class Identifier {
    public abstract field: string = 'default value'; // реализация
допустима
    public abstract get prop(): string; // реализация не допустима
    public abstract set prop(value: string); // реализация не допустима

    public abstract method(): void; // реализация не допустима
}
```

Абстрактный класс, расширяющий другой абстрактный класс, не обязан переопределять все абстрактные члены своего суперкласса. В отличие от абстрактных классов, обычные классы расширяющие абстрактные классы, обязанные переопределить все поля, свойства и методы, находящиеся в иерархической цепочке и помеченные ключевым словом `abstract`, если они не были реализованы предками ранее.

ts

```
abstract class SuperAbstractClass {
    public abstract field: string; // объявление абстрактного поля
```

```
}

abstract class SubAbstractClass extends SuperAbstractClass {} // в
абстрактных потомках допускается не переопределять абстрактные члены
предков

class SubConcreteClass extends SubAbstractClass { // конкретный
подкласс обязан переопределять абстрактные члены, если они...
    public field: string;
}

class SubSubConcreteClass extends SubConcreteClass {} // ... если они
не были переопределены в классах-предках
```

Как было сказано ранее, абстрактным полям может быть задано значение по умолчанию, но в этом случае обратится к нему могут только абстрактные классы в иерархии наследования.

ts

```
abstract class SuperAbstractClass {
    public abstract field0: string = 'default value'; // объявление
абстрактного поля со значением по-умолчанию
    public abstract field1: string;
    public abstract field2: string;
}

abstract class SubAbstractClass extends SuperAbstractClass {
    public field1: string = this.field0; // переопределение
абстрактного поля и инициализация его значением абстрактного поля,
которому было присвоено значение по умолчанию в абстрактном предке
}

class SubOnCreteClass extends SubAbstractClass {
    public field0: string; // конкретному классу необходимо
переопределить два абстрактных поля, так как в предках был
переопределен только один член
    public field2: string;
}
```

Абстрактные члены в полной мере удовлетворяют всем условиям реализации интерфейса. Другими словами, абстрактный класс, декларирующий реализацию интерфейса, может не реализовывать его члены, а лишь пометить их как абстрактные, тем самым переложить реализацию на своих потомков.

ts

```
interface IInterface {
    field: string;
    method(): void;
}
```

```
abstract class AbstractSuperClass implements IInterface { //
абстрактный класс декларирует реализацию интерфейса
    public abstract field: string; // поле без реализации...
    public abstract method(): void; // ...метод без реализации. Тем не
менее ошибки не возникает
}
```

Кроме абстрактных членов, абстрактные классы могут содержать обычные члены, обращение к которым ничем не отличается от членов, объявленных в обычных классах.

Как правило, абстрактные классы реализуют только ту логику, которая не будет ни при каких обстоятельствах противоречить логике своих подклассов.

ts

```
abstract class AbstractSuperClass {
    abstract name: string = "AbstractSuperClass";

    public toString(): string {
        // реализация общего не абстрактного метода
        return `[object ${this.name}]`;
    }
}

class FirstConcreteSubClass extends AbstractSuperClass {
    public name: string = "FirstConcreteSubClass"; // реализуем
абстрактное поле
}

class SecondConcreteSubClass extends AbstractSuperClass {
    public name: string = "SecondConcreteSubClass"; // реализуем
абстрактное поле
}

let first: FirstConcreteSubClass = new FirstConcreteSubClass();
let second: SecondConcreteSubClass = new SecondConcreteSubClass();

first.toString(); // [object FirstConcreteSubClass] реализация в
абстрактном предке
second.toString(); // [object SecondConcreteSubClass] реализация в
абстрактном предке
```

[25.1] Теория

Пришло время разобраться в теории абстрактных классов, а именно ответить на вопросы, которые могут возникнуть при разработке программ.

Интерфейс или абстрактный класс — частый вопрос, ответ на который не всегда очевиден. В действительности это абсолютно разные конструкции, как с точки зрения реализации, так и идеологии. Интерфейсы предназначены для описания публичного *api*, которое служит для сопряжения с программой. Кроме того, они не должны, а в *TypeScript* и не могут, реализовывать бизнес логику той части, которую представляют. Они — идеальные кандидаты для реализации *слабой связанности* (low coupling). При проектировании программ упор должен делаться именно на интерфейсы.

Абстрактные классы, при необходимости, должны реализовывать интерфейсы в той же степени и для тех же целей, что и обычные классы. Их однозначно нужно использовать в качестве базового типа тогда, когда множество логически связанных классов имеет общую для всех логику, использование которой в чистом виде не имеет смысла. Другими словами, если логика размещенная в классе не может или не должна выполнятся отдельно от потомков, то необходимо запретить создание экземпляров подобных классов.

К примеру, абстрактный класс `Animal`, реализующий интерфейс `IAnimal` с двумя членами: свойством `isAlive` и методом `voice`, может и должен реализовать свойство `isAlive`, так как это свойство имеет заранее известное количество состояний (жив или мертв) и не может отличаться в зависимости от потомка. В, то время как метод `voice` (подать голос) как раз таки будет иметь разную реализацию, в зависимости от потомков, ведь коты мяукают, а вороны каркают.

Тем не менее, резонно может возникнуть вопрос, а почему бы не вынести этот функционал в обычный, базовый класс?

Абстрактный класс способен не только подсказать архитектору, что данная сущность является абстрактной для предметной области, то есть не является самостоятельной частью, но также не позволит создать экземпляр класса, работа которого может сломать приложение.

Еще раз тоже самое, но другими словами. Поскольку базовый класс будет реализовывать логику, предполагаемую интерфейсами, разбитыми по принципу разделения интерфейсов, с помощью которых и будет происходить сопряжение с остальными частями программы, то существует возможность попадания его экземпляра в места, предполагающие логику отсутствующую в нем. То есть высокоуровневая логика, присущая только потомкам, может быть сокрыта за менее специфичным интерфейсом, реализуемым самим базовым классом. Чтобы избежать подобных сценариев

допускающих возникновение ошибок во время выполнения, необходимо запретить создание экземпляров подобных классов. (Принцип разделения интерфейсов рассматривается в главе “Типы - Interface”)

Кроме того, абстрактный класс с помощью абстрактных членов не даст разработчику забыть реализовать необходимую логику в потомках.

Но и это ещё не все. Интерфейс `IAnimal` в реальности будет составным типом. То есть, он будет принадлежать к типу `ILiveable`, описывающему свойство `isAlive` и типу `IVoiceable`, описывающему метод `voice`. Реализовать подобное с помощью абстрактного класса не получится, так как класс может расширять только один другой класс, в, то время как интерфейсы могут расширять множество других интерфейсов, и следовательно, принадлежит ко множеству типов данных одновременно. Как раз это и демонстрирует интерфейс `IAnimal` расширяя интерфейсы `ILiveable` и `IVoiceable`.

Ещё часто можно встретить вопрос о замене интерфейсов абстрактными классами. Технически абстрактный класс состоящий только из абстрактных членов, может исполнять роль идеологически отведенную интерфейсу. Но об этом лучше забыть, поскольку для описания открытой части объекта предназначен интерфейс.

Глава 26

Полиморфный тип this

Как в предке описать тип потомка и при чем здесь полиморфный тип `this` - сможет ответить текущая глава.

[26.0] this - как тип

В *TypeScript* существует возможность указывать в качестве типа *полиморфный тип* `this` .

Полиморфный тип — это тип, который представляет множество типов, как единое целое.

Полиморфный тип `this` является совокупностью типов, определяющих тип экземпляра. Кроме того, полиморфный тип `this` доступен для указания только в классах и интерфейсах.

Чтобы понять о чем идет речь, нужно представить два класса, связанных с помощью механизма наследования и в каждом из них объявлен метод, возвращаемое значение которого принадлежит к типу представляющего экземпляр класса, в котором он определен.

ts

```ts
class Animal {
    public sit(): Animal { // реализация метода
        return this;
    }
```

```
}

class Bird extends Animal {
    public fly(): Bird { // дополнение супертипа специфичным методом
        return this;
    }
}
```

Если создать экземпляр подкласса и вызвать по цепочке метод подкласса, а затем — суперкласса, то операция завершится успехом.

ts

```
const bird: Bird = new Bird()
    .fly() // Ok, возвращает тип Bird
    .sit(); // Ok, возвращает тип Animal
```

Если попробовать изменить порядок вызова методов, то возникнет ошибка. Произойдет это по той причине, что метод, объявленный в суперклассе, возвращает значение, принадлежащее к типу самого суперкласса, который ничего не знает о методах, объявленных в его подтипах.

ts

```
const bird: Bird = new Bird()
    .sit() // Ok, возвращает тип Animal
    .fly(); // Error, в типе Animal, возвращенного на предыдущем шаге,
нет объявления метода fly
```

Можно, конечно, в качестве возвращаемого значения указать тип `any`, но помимо того, что пропадет автодополнение, ещё и пострадает семантическая привлекательность кода.

ts

```
class Animal {
    public sit(): any {
        return this;
    }
}

class Bird extends Animal {
    public fly(): any {
        return this;
    }
}

let bird: Bird = new Bird()
    .fly() // Ok
    .sit(); // Ok

bird = new Bird()
```

```
    .sit() // Ok
    .fly(); // Ok, работает, но так лучше не делать
```

TypeScript предлагает решить эту проблему с помощью полиморфного типа `this`. Ожидаемое поведение достигается за счет того, что полиморфный тип `this` является множеством типов, определяемого цепочкой наследования. Другими словами, тип `this` будет принадлежать к тому же типу, что и экземпляр подкласса, который принадлежит к типу подкласса и типу суперкласса одновременно.

Такое поведение называется *F-ограниченный полиморфизм* (F-bounded polymorphism).

ts

```ts
class Animal {
    public sit(): this {
        return this;
    }
}

class Bird extends Animal {
    public fly(): this {
        return this;
    }
}

let bird = new Bird()
    .fly() // Ok
    .sit(); // Ok

bird = new Bird()
    .sit() // Ok, возвращает тип Bird
    .fly(); // Ok
```

Стоит отдельно подчеркнуть, что полиморфный тип `this` не принадлежит к типу класса или интерфейса в котором указан. Полиморфный тип `this` может быть определен только на основе экземпляра класса. Проще говоря, полиморфный тип `this` принадлежит к типу своего экземпляра и может быть определен только в момент создания экземпляра. Так же тип `this` совместим с типом `any`, а при условии, что флаг `--strictNullChecks` установлен в `false`, ещё и к типам `null` и `undefined`. К тому же тип `this` совместим с типом экземпляра, ссылку на который можно получить с помощью ключевого слова `this`.

ts

```ts
class Animal {
    public animalAll: this[] = []; // массив с полиморфным типом this

    constructor() {
        this.add(new Animal()); // Error, так как на данном этапе не
известно, к какому типу будет принадлежать полиморфный тип this
        this.add(this); // Ok
```

```
    }

    public add(animal: this): this {
        this.animalAll.push(animal);

        return this;
    }
}

class Type {
    static interface: typeof Animal = Animal;
    static animal: Animal = new Animal();
    static any: any = new Animal();
    static null: null = null;
    static undefined: undefined = undefined;
}

const animal = new Animal()
    .add(Type.animal) // Ok
    .add(Type.interface) // Error
    .add(Type.any) // Ok
    .add(Type.null) // Ok
    .add(Type.undefined); // Ok
```

Не будет лишним упомянуть, что в тех случаях, когда тип не указан явно, а в качестве значения выступает ссылка на экземпляр (`this`), то вывод типов будет указывать на принадлежность к полиморфному типу `this` .

ts

```
class Animal {
    public instance = this; // instance: this

    public getInstance() { // getInstance(): this
        const instance = this; // instance: this

        return this;
    }
}
```

Глава 27

Модификатор readonly (только для чтения)

Локальную переменную можно определить *неизменяемой* при помощи ключевого слова `const`. Но как реализовать подобное поведение для полей класса? На данный вопрос как раз и призвана ответить текущая глава.

[27.0] Модификатор readonly

В *TypeScript* существует модификатор, с помощью которого поле объекта помечается как *“только для чтения”*. Модификатор, запрещающий изменение значений полей объектов, указывается с помощью ключевого слова `readonly`. Он применяется к полям как при их объявлении в классе, так и при их описании в интерфейсе.

ts

```ts
interface IAnimal {
    readonly name: string;
}

class Animal {
    public readonly name: string = 'name';
}
```

Если при описании интерфейса поле было помечено как `readonly`, то классам, реализующим этот интерфейс, необязательно указывать модификатор `readonly`. Но в таком случае значение поля будет изменяемым.

ts

```ts
interface IAnimal {
    readonly name: string;
}

class Bird implements IAnimal {
    public readonly name: string = 'bird'; // Ok
}

class Fish implements IAnimal {
    public name: string = 'fish'; // Ok
}

const bird: Bird = new Bird();
bird.name = 'newBird'; // Error

const fish: Fish = new Fish();
fish.name = 'newFish'; // Ok
```

Это правило работает и в обратном направлении — поле, описанное в интерфейсе без указания модификатора `readonly`, может быть помечено этим модификатором при реализации.

ts

```ts
interface IAnimal {
    name: string;
}

class Bird implements IAnimal {
    public readonly name: string = 'bird'; // Ok
}

const bird: Bird = new Bird();
bird.name = 'newBird'; // Error
```

Модификатор `readonly`, примененный к параметрам конструктора, заставляет компилятор расценивать их как поля класса.

ts

```ts
interface IAnimal {
    name: string;
}

class Bird implements IAnimal {
```

```
    constructor(readonly name: string) {}
}

const bird: Bird = new Bird('bird');
```

Кроме того, модификатор `readonly`, примененный к параметрам конструктора, можно комбинировать с другими модификаторами доступа.

ts

```ts
interface IAnimal {
    name: string;
}

class Bird implements IAnimal {
    constructor(private readonly name: string) {}
}

const bird: Bird = new Bird('bird');
```

Полю, к которому применен модификатор `readonly`, необязательно присваивать значение в момент объявления. Но в таком случае присвоить ему значение можно будет только из конструктора класса, в котором это поле объявлено. Если полю был применён модификатор `readonly` и не было присвоено значение, то такое поле, так же как и любое другое неинициализированное поле, будет иметь значение `undefined`.

ts

```ts
interface IAnimal {
    readonly name: string;
}

class Animal implements IAnimal {
    public readonly name: string; // Ok

    constructor(){
        this.name = 'animal'; // Ok
    }
}
```

Попытка присвоить значение полю, к которому применен модификатор `readonly`, в месте, отличном от места объявления или конструктора класса, приведет к возникновению ошибки.

ts

```ts
interface IAnimal {
    readonly name: string;
}
```

```
class Animal implements IAnimal {
    public readonly name: string; // Ok

    public setName(name: string): void {
        this.name = name; // Error
    }
}
```

Не получится избежать возникновения ошибки и при попытке присвоить значение из конструктора класса-потомка (с условием, что потомок не переопределяет его).

ts

```
interface IAnimal {
    readonly name: string;
    readonly age: number;
}

class Animal implements IAnimal {
    public readonly name: string; // Ok
    public readonly age: number; // Ok
}

class Bird extends Animal {
    public readonly age: number; // переопределение поля

    constructor(){
        super();
        super.name = 'bird'; // Error
        this.age = 0; // Ok
    }
}
```

Поле объекта, созданного с помощью литерала объекта, будет невозможно изменить, если в связанном с ним типе к этим полям применен модификатор `readonly` .

ts

```
interface IAnimal {
    readonly name: string;
}

const animal: IAnimal = { name: 'animal' };

animal.name = 'newAnimal'; // Error
```

Если полям, помеченным *“только для чтения”*, не указан тип, а присвоение примитивного значения происходит в месте объявления, то для таких полей вывод типов укажет принадлежность к литеральному типу.

ts

```ts
class Animal {
    public readonly nameReadonly = 'animal'; // nameReadonly: "animal"
    public nameDefault = 'animal';           // nameDefault: string

    public readonly ageReadonly = 0; // ageReadonly: 0
    public ageDefault = 0;           // ageReadonly: number

    public readonly isLifeReadonly = true; // isLifeReadonly: true
    public isLifeDefault = true;           // isLifeDefault: boolean
}
```

Глава 28

Definite Assignment Assertion Modifier

Поля класса или локальные переменные, которые по сценарию могут иметь значение `undefined`, способны обрушить ожидаемый ход программы. Неудивительно, что *TypeScript* реализует механизмы, защищающие от подобных случаев. Но, кроме того, он также реализует механизмы, позволяющие обходить предыдущие, когда поведение лишь похоже на нежелательное по архитектурным причинам. Если кажется не понятным, то не беда, ведь данная глава и призвана помочь в этом разобраться.

[28.0] Модификатор утверждения не принадлежности значения к типу undefined

Для повышения типобезопасности программы, рекомендуется вести разработку с активной опцией `--strict` (глава "Опции компилятора"), активирующей множество других опций изменяющих поведение компилятора и тем самым заставляя разработчиков писать код, сводящий к минимуму ошибки на этапе выполнения.

Это привело к созданию опции `--strictPropertyInitialization`, которая, при активной опции `--strictNullChecks`, запрещает классу иметь поля, типы которых явно не принадлежат к `undefined` и которые не были инициализированы в момент его создания. Таким образом предотвращается обращение к полям которые могут иметь значение `undefined`.

ts

```ts
class Identifier {
    public a: number = 0; // Ok, инициализация при объявлении
    public b: number; // Ok, инициализация в конструкторе
    public c: number | undefined; // Ok, явное указание принадлежности
к типу Undefined
    public d: number; // Error, инициализация отсутствует

    constructor() {
        this.b = 0;
    }
}
```

Но бывают случаи, при которых условия, устанавливаемые опцией `--strictPropertyInitialization`, не могут быть удовлетворены в полной мере. К самым распространенным случаям можно отнести установку значений полей с помощью *DI* (dependency injection), инициализации, вынесенной в методы жизненного цикла (*life cycle*), а также методы инициализации выполняемые из конструктора класса.

ts

```ts
// инициализация с помощью DI
class A {
    @Inject(Symbol.for('key'))
    public field: number; // Error
}
```

ts

```ts
// метод жизненного цикла из Angular
class B {
    private field: number; // Error

    public ngOnInit(): void {
        this.field = 0;
    }
}
```

ts

```ts
// инициализация вне конструктора
class C {
    private field: number; // Error

    constructor(){
        this.init();
    }

    private init(): void {
        this.field = 0;
```

```
    }
}
```

Для таких случаев синтаксис *TypeScript* содержит модификатор *definite assignment assertion modifier*, который указывается с помощью символа восклицательного знака (!), располагаемого после идентификатора поля или переменной.

ts

```ts
class Identifier {
    public identifier!: Type;
}

// или

const identifier!: Type;
```

Применяя модификатор *definite assignment assertion modifier*, разработчик сообщает компилятору, что берет ответственность за инициализацию поля на себя.

ts

```ts
// инициализация с помощью DI
class A {
    @Inject(Symbol.for('key'))
    public field!: number; // Ok
}
```

ts

```ts
// метод жизненного цикла из Angular
class B {
    private field!: number; // Ok

    public ngOnInit(): void {
        this.field = 0;
    }
}
```

ts

```ts
// инициализация вне конструктора
class C {
    private field!: number; // Ok

    constructor(){
        this.init();
    }


    private init(): void {
        this.field = 0;
```

```
    }
}
```

Данный модификатор можно применять только тогда, когда доподлинно известно, что поле или переменная будет инициализирована каким-либо механизмом, выполняемым во время выполнения. В остальных случаях данный механизм лишь сведет на нет работу компилятора *TypeScript*.

Глава 29

Модификатор override

Механизм наследования, особенно при написании так называемых библиотек, позволяет реализовать большую часть необходимого функционала в суперклассах, что значительно сокращает время разработки. Но при работе с наследованием можно встретить острые углы, сгладить которые в *TypeScript* предполагается за счёт оператора `override` опции компилятора `--noImplicitOverride`, которым и будет посвящена эта глава.

[29.0] Модификатор override и флаг --noImplicitOverride

Представьте случай переопределения подклассом некоторых методов своего суперкласса.

ts

```ts
class SuperClass {
    /**
     * [*] Определяет метод
     */
    a(){} // [*]
    b(){}// [*]
}
class SubClass extends SuperClass {
    /**
```

```
     * [*] Переопределяет методы своего суперкласса.
     */
    a(){} // [*]
    b(){} // [*]
}
```

Всё банально просто! Но что, если над проектом работает большое количество команд находящихся в разных уголках земного шара и команде занимающейся разработкой `SuperClass` неожиданно придеёт в голову изменить его *api* удалив оба метода? В таком случае, разработчики класса `SubClass` даже не узнают об этом, поскольку *переопределение* превратится в *определение*. Другими словами, компилятор даже глазом не моргнет, поскольку посчитает, что класс `SubClass` определят методы `a()` и `b()`.

ts

```
class SuperClass {
    /**
     * Удалили a() и b() и добавили c().
     */
    c(){}
}
class SubClass extends SuperClass {
    /**
     * Ошибки не возникает, так как компилятор считает
     * что данный класс определяет оба метода.
     */
    a(){}
    b(){}
}
```

Для предотвращения подобных сценариев, в *TypeScript* существует модификатор `override`, который однозначно указывает на переопределение методов родительского класса. При использовании модификатора `override` компилятор сможет понять, что происходит переопределение несуществующих в суперклассе методов и сообщить об этом с помощью ошибки.

ts

```
class SuperClass {
    /**
     * Удалили a() и b() и добавили c().
     */
    c(){}
}
class SubClass extends SuperClass {
    /**
     * [*] Error ->
     * This member cannot have an 'override' modifier
     * because it is not declared in the base class
'SuperClass'.ts(4113)
     *
```

```
     * Теперь компилятор понимает, что происходи переопределение
     * несуществующих методов.
     */
    override a(){} // [*]
    override b(){} // [*]
}
```

Также при использовании механизма наследования может возникнуть диаметрально противоположная ситуация, при которой суперкласс может объявить метод, который уже существует в его потомке.

ts

```
/**
 * [0] метод определенный только в SubClass
 */
class SuperClass {
    a(){}
}
class SubClass extends SuperClass {
    b(){} // [0]
}

/**
 * [1] Но спустя некоторое время класс SuperClass
 * определяет метод b(), который уже существует в
 * классе-потомке [2]. Другими словами, произошло
 * нежелаемое переопределение способное привести
 * к непредсказуемому поведению программы.
 */
class SuperClass {
    a(){}
    b(){} // [1]
}
class SubClass extends SuperClass {
    b(){} // [2]
}
```

Предотвратить нежелательное поведение в *TypeScript* можно с помощью флага `--noImplicitOverride`, при активации которого, в подобных случаях будет возникать ошибка.

ts

```
class SuperClass {
    a(){}
    b(){}
}
class SubClass extends SuperClass {
    /**
```

```
     * --noImplicitOverride = true
     *
     * [*] Error -> This member must have an 'override'
     * modifier because it overrides a member in the base
     * class 'SuperClass'.ts(4114)
     */
    b(){} // [*]
}
```

Глава 30

Классы — Тонкости

В *TypeScript* с классами связанно несколько неочевидных моментов. Неожиданная встреча с ними на практике непременно приведет к возникновению множества вопросов, ответы на который содержит текущая глава.

[30.0] Классы - Тонкости implements

Кроме того, что класс может реализовать (`implements`) интерфейсы (`interface`), он также может реализовать другой класс.

ts

```ts
interface IAnimal {
    name: string;
}

class Animal {
    public name: string;
}

class Bird implements IAnimal { // Ok
    public name: string;
}
class Fish implements Animal { // Ok
    public name: string;
}
```

Как уже можно было догадаться, при реализации классом другого класса действуют те же правила, что и при расширении класса интерфейсом. То есть класс, у которого все члены объявлены как публичные (`public`), может реализовать любой другой класс. Если класс имеет определение членов с модификаторами доступа `private` или `protected` , то его может реализовать только этот же класс или его потомки.

ts

```
class Animal {
    public name: string;
}

class Bird implements Animal { // Ok
    public name: string;
    protected age: number;
}

class Fish implements Animal { // Ok
    public name: string;
    private arial: string;
}

class Raven implements Bird { // Error
    public name: string;
    protected age: number;
}

class Owl extends Bird implements Bird { // Ok
    public name: string;
    protected age: number;
}

class Shark implements Fish { // Error
    public name: string;
}

class Barracuda extends Fish implements Fish { // Ok
    public name: string;
}
```

[30.1] Частичное слияние интерфейса с классом

На текущий момент известно, что два интерфейса, объявленные в одной области видимости, сливаются вместе. Кроме этого, если интерфейс объявлен в одной области

видимости с одноимённым классом, то компилятор считает, что класс реализовал этот интерфейс.

ts

```ts
interface Animal {
    id: string;
    age: number;
}

class Animal {}

const animal = new Animal(); // Ok

animal.id = 'animal'; // Ok
animal.age = 0; // Ok

const { id, age } = animal; // Ok -> id: string and age: number

console.log(id, age); // 'animal', 0
```

[30.2] Переопределение свойств полями и наоборот при наследовании

В *JavaScript* при использовании механизма наследования (`extends`) производный класс в состоянии переопределить свойство объявленное в базовом классе полем и наоборот, поле свойством.

js

```js
class Base {
    get value(){
        return 'base'
    }
    set value(value){
        console.log(value);
    }
}

class Derived extends Base {
    value = 'derived'
}

let derived = new Derived();
```

```
console.log(derived.value); // 'derived'

derived.value = `new derived`; // не сложно догадаться, что при
присваивании нового значения console.log в сеттер базового класса
вызвана не будет

console.log(derived.value); // 'new derived'

/**
 * Тоже справедливо и для переопределения
 * поля объявленного в базовом классе свойствами
 * производного класса.
 */
```

Но, во избежание казусов сопряженных с этим поведением, *TypeScript* запрещает переопределения при наследовании.

ts

```
class Base {
    get value() {
        return 'value';
    }
    set value(value: string) {

    }
}

class Derived extends Base {
    /**
     * Error ->
     *
     * 'value' is defined as an accessor in class 'Base',
     * but is overridden here in 'Derived'
     * as an instance property.
     */
    value = 'value';
}
```

ts

```
class Base {
     value = 'value';
}

class Derived extends Base {
    /**
     * Error ->
     *
     * 'value' is defined as a property in class 'Base',
     * but is overridden here in 'Derived' as an accessor.
     */
```

```
    get value() {
        return 'value';
    }
    set value(value: string) {

    }
}
```

Глава 31

Различия var, let, const и модификатора readonly при неявном указании примитивных типов

После того, как было рассмотрено понятие *литеральных типов*, и тем самым подведена черта под примитивными типами, наступило время рассмотреть особенности `var`, `let` и `const`, определение которых не содержит явную аннотацию типа.

[31.0] Нюансы на практике

Отсутствие аннотации типа у объявления переменной означает, что тип будет определен с помощью механизма вывода типов. В тот момент, когда `var` или `let` объявляется, но не инициализируется, вывод типов определяет их принадлежность к типу `any`. В случае с `const` невозможно обойтись без инициализации в момент объявления.

ts

```ts
var a; // a: any
let b; // b: any
const c; // Error
```

Если при определении `var` или `let` аннотация типа была опущена, а присваиваемое значение принадлежит к примитивному типу, то вывод типа определит принадлежность на основе типа значения. Для цифр, это будет тип `number`, для строк `string` и т.д.

ts

```ts
var a = 5; // a: number
let b = 'text'; // b: string
```

Но при аналогичной ситуации для `const`, вывод типов определит принадлежность не к типу значения, а к литеральному типу представляемого этим значением.

ts

```ts
const a = 5; // a: 5
const b = 'text'; b: 'text'
```

Поведение вывода типов для поля, объявленного с модификатором `readonly`, аналогично объявлению `const`, то есть для примитивов будет выведен литеральный тип.

ts

```ts
class T1 {
    readonly f1 = 'text'; // f1: text
    readonly f2 = 0; // f2: 0
    readonly f3 = true; // f3: true
}
```

Более подробно эта тема рассматривается далее в главе “Вывод типов”.

Глава 32

Аксессоры

Аксесоры или геттеры и сеттеры, или привычнее — свойства, незаменимая часть языка *JavaScript*, без которой сложно представить современную разработку. Но начинающим *TypeScript* разработчикам могут быть не совсем очевидны некоторые моменты, каждый из которых будет подробно рассмотрен в этой главе.

[32.0] Отдельные типы аксессоров

На практике, могут возникнуть случаи, когда сеттер нуждается в установке значения, тип которого отличен от типа возвращаемого геттером. И *TypeScript* позволяет реализовать это.

ts

```ts
/**
 * Геттер возвращает тип number, в то время, как
 * сеттер ожидает значение принадлежащие к типу объединению
 * number | string | boolean
 */
class T0 {
    private _value = 0;

    get value(): number {
        return this._value;
    }
    set value(value: number | string | boolean){
```

```
    }
}
```

Но есть один не очевидный момент заключающийся в том, что определение типа значения сеттера обязанно включать тип, который возвращает геттер. В противном случае возникнет ошибка.

js

```
class T {
    private _value = 0;

    /**
     * [*] Error -> The return type of a 'get' accessor must be
assignable to its 'set' accessor typets(2380)
     */
    get value(): number { // [*]
        return this._value;
    }

    /**
     * Сеттер не включает тип возвращаемый геттером.
     */
    set value(value: string | boolean){
    }
}
```

И поскольку аксессоры могут быть объявлены и в объекте созданного при помощи литерала объекта ({}), стоит также упомянуть, что правила для них ничем не отличаются от правил для классов.

Глава 33

Операторы - Optional, Not-Null Not-Undefined, Definite Assignment Assertion

Оператор `Optional`, помечающий члены и параметры как необязательные, довольно часто используется при разработке приложений. И если в понимании механизма его работы нет ничего сложного, то для идеологически связанного с ним оператора `Not-Null Not-Undefined` не все так очевидно.

[33.0] Необязательные поля, параметры и методы (Optional Fields, Parameters and Methods)

В *TypeScript* существует возможность декларировать поля, методы и параметры как *необязательные*. Эта возможность позволяет исключать помеченные элементы из инициализации, вызовов и проверки на совместимость.

Поле, параметр или метод помечается как *необязательный* с помощью оператора вопросительного знака `?`. При объявлении полей и параметров, оператор помещается сразу после идентификатора `identifier?: Type`. Для методов оператор помещается между идентификатором и круглыми скобками `identifier?(): Type`.

ts

```ts
type VoiceEvent = {
    type: string
    repeat?: number // необязательное поле
};

type VoiceHandler = (event?: VoiceEvent) => void; // необязательный
параметр функции

class Animal {
   name?: number; // необязательное поле

   voice?(): void {} // необязательный метод
}
```

Термины *поля*, *параметры* и *методы* делают данный оператор чересчур именитым. Поэтому в дальнейшем он будет упрощен до *“необязательного оператора”*.

Из темы посвященной типу `undefined` стало известно, что он является подтипом всех типов. Это в свою очередь означает, что его единственное значение - `undefined` - можно присвоить в качестве значения любому другому типу.

ts

```ts
/** strictNullChecks: false */

let a: number = undefined; // Ok
let b: string = undefined; // Ok
let c: boolean = undefined; // Ok
let d: object = undefined; // Ok
let e: any = undefined; // Ok
```

Когда у компилятора флаг `--strictNullChecks` установлен в `true`, тип `undefined` является подтипом только типа `any`. Это означает, что связать значение `undefined` можно только с типом `any`.

ts

```ts
/** strictNullChecks: true */

let a: number = undefined; // Error
let b: string = undefined; // Error
let c: boolean = undefined; // Error
let d: object = undefined; // Error
let e: any = undefined; // Ok
```

Как было сказано в начале, *необязательное* буквально означает, что параметр функции может быть не ассоциирован со значением, а поле или метод и вовсе не существовать в объекте. А как известно, неинициализированные члены объектов и параметры функции всегда принадлежат к типу `undefined`. Поэтому каждый раз, когда компилятор видит

поля или параметры, помеченные как необязательные, он расценивает это как явное указание на сценарий, допускающий значение `undefined`, способное нарушить ожидаемый ход выполнения программы. И поскольку активация рекомендуемого флага `--strictNullChecks` запрещает присваивать значение `undefined` типам отличным от `undefined` или `any`, вывод типов берет на себя инициативу и помечает все необязательные конструкции как принадлежащие к объединению, включающее тип `undefined`.

ts

```ts
/** strictNullChecks: true */

let a: { field?: number }; // field: number | undefined
let b: { field?: string }; // field: string | undefined
let c: { field?: boolean }; // field: boolean | undefined
let d: (prop?: object) => void; // prop: object | undefined
let e: (prop?: any) => void; // prop: any
let f: (prop?: number | undefined) => void; // prop: number | undefined
```

Когда флаг `--strictNullChecks` установлен в `false` и он встречает поля или параметры, помеченные как необязательные, он точно также понимает, что по сценарию допускается значение `undefined`. Но при этом он не добавляет к уже указанному типу тип `undefined` и даже не берет его в расчет при явном указании. Такое поведение связано с тем, что при неактивном флаге `--strictNullChecks`, тип данных `undefined` совместим со всеми остальными типами. Это, в свою очередь, освобождает поля и параметры от его явного указания.

ts

```ts
/** strictNullChecks: false */

let a: { field?: number }; // field: number
let b: { field?: string }; // field: string
let c: { field?: boolean }; // field: boolean
let d: (prop?: object) => void; // prop: object
let e: (prop?: any) => void; // prop: any
let f: (prop?: number | undefined) => void; // prop: number
```

Также стоит упомянуть, что необязательные поля необязательно должны содержать явную аннотацию.

ts

```
interface IT1 {
    f1?; // Ok -> f1?: any
}

class T1 {
    f1?; // Ok -> f1?: any
    f2? = 0; // Ok -> f2?: number
}
```

Поскольку значение `undefined` присвоенное полю объекта далеко не то же самое, что отсутствие члена вовсе, при котором также возвращается `undefined`, в *TypeScript* существует флаг `--exactOptionalPropertyTypes`, при активации которого, в подобных случаях будут возникать ошибки.

ts

```
type T = {
    a: number;
    b?: string;
}

let o: T = {
    a: 5,
    b: undefined // Error -> Type 'undefined' is not assignable to type
'string'.ts(2322)
};
```

[33.1] Оператор ! (Non-Null and Non-Undefined Operator)

Оператор `Not-Null Not-Undefined`, при активной опции `--strictNullChecks`, в случаях, допускающих обращение к несуществующим членам, позволяет приглушать сообщения об ошибках.

Простыми словами, когда в режиме `--strictNullChecks` происходит обращение к значению объекта или метода, которые могут иметь значение `null` или `undefined`, компилятор, с целью предотвращения возможной ошибки, накладывает запрет на операции обращения и вызова. Разрешить подобные операции возможно с помощью оператора `Not-Null Not-Undefined`, который обозначается восклицательным знаком `!`.

Чтобы понять принцип оператора `Non-Null Non-Undefined`, достаточно представить слушатель события, у которого единственный параметр `event`, принадлежность которого указана к типу `UserEvent`, помечен как необязательный. Это означает, что помимо обусловленного типа `UserEvent`, параметр может принадлежать ещё и к типу `undefined`. А это значит, что при попытке обратится к какому-либо члену объекта события `event`, может возникнуть исключение, вызванное обращением через ссылку на `null` или `undefined`. С целью предотвращения исключения во время выполнения, компилятор, во время компиляции, выведет сообщение об ошибке, вызванной обнаружением потенциально опасного кода.

ts

```ts
/** strictNullChecks: true */

type UserEvent = { type: string };

// параметр помечен как необязательный, поэтому тип выводится как
event?: UserEvent | undefined
function handler(event?: UserEvent): void {
    // потенциальная ошибка, возможно обращение к полю несуществующего
объекта
    let type = event.type; // Error -> возможная ошибка во время
выполнения
}
```

Обычно в таких случаях стоит изменить архитектуру, но если разработчик в полной мере осознает последствия, то компилятор можно настоятельно попросить закрыть глаза на потенциально опасное место при помощи оператора `Not-Null Not-Undefined`. При обращении к полям и свойствам объекта, оператор `Not-Null Not-Undefined` указывается перед оператором точка `object!.field`.

ts

```ts
/** strictNullChecks: true  */

type UserEvent = { type: string };

function handler(event?: UserEvent): void {
    // указываем компилятору, что берем этот участок кода под
собственный контроль
    let type = event!.type; // Ok
}
```

Оператор `Not-Null Not-Undefined` нужно повторять каждый раз, когда происходит обращение к полям и свойствам объекта, помеченного как необязательный.

ts

```ts
/** strictNullChecks: true  */

type Target = { name: string };
```

```
type CurrentTarget = { name };

type UserEvent = {
    type: string,
    target?: Target,
    currentTarget: CurrentTarget
};

function handler(event?: UserEvent): void {
    let type = event!.type; // 1 !
    let target = event!.target!.name; // 2 !
    let currentTarget = event!.currentTarget.name; // 1 !
}
```

При обращении к необязательным методам объекта, оператор `Not-Null Not-Undefined` указывается между идентификатором (именем) и круглыми скобками. Стоит обратить внимание, что когда происходит обращение к необязательному полю или свойству объекта, оператор `Not-Null Not-Undefined` указывается лишь один раз `optionalObject!.firstLevel.secondLevel`. При обращении к необязательному методу того же объекта, оператор `Not-Null Not-Undefined` указывается дважды `optionalObject!.toString!()`.

ts

```
/** strictNullChecks: true  */

type Target = { name: string };

type CurrentTarget = { name };

type UserEvent = {
    type: string,
    target?: Target,
    currentTarget: CurrentTarget,
    toString?(): string
};

function handler(event?: UserEvent): void {
    let type = event!.type; // 1 !
    let target = event!.target!.name; // 2 !
    let currentTarget = event!.currentTarget.name; // 1 !
    let meta = event!.toString!(); // 2 !
}
```

Нужно повторить ещё раз, что оператор `Not-Null Not-Undefined`, при активном флаге `--strictNullChecks`, обязателен только в случаях, когда объект принадлежит к типу отличного от `any`.

ts

```ts
/** strictNullChecks: true  */

type Target = { name: string };

type CurrentTarget = { name };

type UserEvent = {
    type: string,
    target?: Target,
    currentTarget: CurrentTarget,
    toString?(): string,
    valueOf(): any
};

function handler(event?: any): void {
    let type = event.type; // 0 !
    let target = event.target.name; // 0 !
    let currentTarget = event.currentTarget.name; // 0 !
    let meta = event.toString(); // 0 !
    let value = event.valueOf(); // 0 !
}
```

И, как было сказано в самом начале, правило оператора `Not-Null Not-Undefined`, применённое к необязательному оператору, идентично для всех полей и параметров, принадлежащих к типам `null` или `undefined` ...

ts

```ts
/** strictNullChecks: true  */

type Target = { name: string };

type CurrentTarget = { name };

type UserEvent = {
    type: string,
    target?: Target,
    currentTarget: CurrentTarget,
    toString?(): string,
    valueOf(): any
};

function handler(event: UserEvent | undefined): void {
    let type = event.type; // Error
    let target = event.target.name; // Error
    let currentTarget = event.currentTarget.name; // Error
    let meta = event.toString(); // Error
    let value = event.valueOf(); // Error
}
```

...при условии, что они не будут принадлежать к типу `any`.

ts

```ts
/** strictNullChecks: true  */

type Target = { name: string };

type CurrentTarget = { name };

type UserEvent = {
    type: string,
    target?: Target,
    currentTarget: CurrentTarget,
    toString?(): string,
    valueOf(): any
};

function handler(event: UserEvent | undefined | any): void {
    let type = event.type; // Ok
    let target = event.target.name; // Ok
    let currentTarget = event.currentTarget.name; // Ok
    let meta = event.toString(); // Ok
    let value = event.valueOf(); // Ok
}
```

[33.2] Оператор ! (Definite Assignment Assertion)

Для повышения типобезопасности программы, правила, накладываемые опцией `--strictNullChecks` (глава "Опции компилятора"), действуют также на переменные, инициализирующиеся в чужом контексте.

ts

```ts
let value: number;

initialize();

console.log(value + value); // Error, обращение к переменной перед присвоением ей значения

function initialize() {
  value = 0;
}
```

Чтобы избежать ошибки при обращении к переменным, которые инициализированы в чужом контексте, нужно использовать *definite assignment assertions*. *Definite assignment assertions* также указывается с помощью символа восклицательного знака (!) и располагается после идентификатора переменной. Указывая данный оператор каждый раз при обращении к переменной, разработчик сообщает компилятору, что берет на себя все проблемы, которые могут быть вызваны отсутствием значения у переменной.

ts

```ts
let value: number;

initialize();

console.log(value! + value!); // Ok, указание definite assignment
assertion

function initialize() {
  value = 0;
}
```

Глава 34

Обобщения (Generics)

Из всего, что стало и ещё станет известным о типизированном мире, тем, кто только начинает свое знакомство с ним, тема, посвященная обобщениям (*generics*), может казаться наиболее сложной. Хотя данная тема, как и все остальные, обладает некоторыми нюансами, каждый из которых будет детально разобран, в реальности, рассматриваемые в ней механизмы очень просты и схватываются на лету. Поэтому приготовьтесь, к концу главы место занимаемое множеством вопросов, касающихся обобщений, займет желание сделать все пользовательские конструкции универсальными.

[34.0] Обобщения - общие понятия

Представьте огромный, дорогущий и высокотехнологичный типографский печатный станок, выполненный в виде монолита, что в свою очередь делает его пригодным для печати только одного номера газеты. То есть для печати сегодняшних новостей необходим один печатный станок, для завтрашних, другой и т.д. Подобный станок сравним с *обычным типом* признаки которого после объявления остаются неизменны при его реализации. Другими словами, если при существовании типа `A`, описание которого включает поле, принадлежащее к типу `number`, потребуется тип, отличие которого будет состоять лишь в принадлежности поля к другому типу, возникнет необходимость в его объявлении.

ts

```ts
// простые типы сравнимы с монолитами
```

```
// этот станок предназначен для печати газет под номером A
interface A {
    field: number;
}

// этот станок предназначен для печати газет под номером B
interface B {
    field: string;
}

// и т.д.
```

К счастью, в нашей реальности нашли решение не только относительно печатных станков, но и типов. Нежелания тратить усилия на постоянное описывание монолитных типов послужило причиной зарождения парадигмы *обобщенного программирования*.

Обобщенное программирование (*Generic Programming*) — это подход, при котором алгоритмы могут одинаково работать с данными, принадлежащими к разным типам данных без изменения декларации (описания типа).

В основе обобщенного программирования лежит такое ключевое понятие как *обобщение* . *Обобщение* (*Generics*) — это *параметризированный тип* позволяющий объявлять *параметры типа*, являющиеся временной заменой *конкретных типов*, *конкретизация* которых будет выполнена в момент создания экземпляра. Параметры типа, при условии соблюдения некоторых правил, можно использовать в большинстве операций, допускающих работу с обычными типами. Все это вместе дает повод сравнивать обобщенный тип с *правильной версией* печатного станка, чьи заменяемые валы, предназначенные для отпечатывания информации на проходящей через них бумаге, сопоставимы с *параметрами типа*.

В реальности обобщения позволяют сокращать количество преобразований (приведений) и писать многократно используемый код, при этом повышая его типобезопасность.

Этих примеров должно быть достаточно для образования отчетливого образа обобщений. Но прежде чем продолжить, стоит уточнить значения таких далеко не всем очевидных терминов, как: *обобщенный тип*, *параметризированный тип* и *универсальная конструкция*.

Для понимания этих терминов необходимо представить чертеж бумажного домика, в который планируется поселить пойманного на пикнике жука. Когда гипотетический жук мысленно располагается вне границ начерченного жилища, сопоставимого с типом, то оно предстает в виде *обобщенного типа*. Когда жук представляется внутри своей будущей обители, то о ней говорят как о *параметризированном типе*. Если же чертеж материализовался, хотя и в форму представленную обычной коробкой из-под печенья, то её называют *универсальной конструкцией*.

Другими словами, тип, определяющий параметр, обозначается как обобщенный тип. При обсуждении типов, представляемых параметрами типа, необходимо понимать, что они определены в параметризированном типе. Когда объявление обобщенного типа

получило реализацию, то такую конструкцию, будь, то класс или функция, называют универсальной (универсальный класс, универсальная функция или метод).

[34.1] Обобщения в TypeScript

В *TypeScript* обобщения могут быть указаны для типов, определяемых с помощью:

- *псевдонимов* (`type`)
- *интерфейсов*, объявленных с помощью ключевого слова `interface`
- *классов* (`class`), в том числе *классовых выражений* (*class expression*)
- *функций* (`function`) определенных в виде как деклараций (*Function Declaration*), так и выражений (*Function Expression*)
- *методов* (*method*)

Обобщения объявляются при помощи пары угловых скобок, в которые через запятую, заключены *параметры типа*, называемые также *типо-заполнителями* или *универсальными параметрами* `Type<T0, T1>` .

ts

```ts
/** [0]  [1] [2] */
interface Type<T0, T1> {}

/**
 * [0] объявление обобщенного типа Type,
 * определяющего два параметра типа [1][2]
 */
```

Параметры типа могут быть указаны в качестве типа везде, где требуется аннотация типа, за исключением членов класса (static members). Область видимости параметров типа ограничена областью обобщенного типа. Все вхождения параметров типа будут заменены на конкретные типы переданные в качестве аргументов типа. Аргументы типа указываются в угловых скобках, в которых через запятую указываются конкретные типы данных `Type<number, string>` .

ts

```ts
/** [0]  [1]      [2] */
let value: Type<number, string>
```

```
/**
 * [0] указание обобщенного типа,
 * которому в качестве аргументов
 * указываются конкретные типы
 * number [1] и string [2]
 */
```

Идентификаторы параметров типа должны начинаться с заглавной буквы и кроме фантазии разработчика они также ограничены общими для *TypeScript* правилами. Если логическую принадлежность параметра типа возможно установить без какого-либо труда, как например в случае `Array<T>`, кричащего, что параметр типа `T` представляет тип, к которым могут принадлежать элементы этого массива, то идентификаторы параметров типа принято выбирать из последовательности `T`, `S`, `U`, `V` и т.д. Также частая последовательность `T`, `U`, `V`, `S` и т.д.

С помощью `K` и `V` принято обозначать типы соответствующие `Key` / `Value`, а при помощи `P` — `Property`. Идентификатором `Z` принято обозначать полиморфный тип `this`.

Кроме того, не исключены случаи, в которых предпочтительнее выглядят полные имена, как, например, `RequestService`, `ResponseService`, к которым ещё можно применить *Венгерскую нотацию* - `TRequestService`, `TResponseService`.

К примеру, увидев в автодополнении редактора тип `Array<T>`, в голову тут же приходит верный вариант, что массив будет содержать элементы принадлежащие к указанному типу `T`. Но, увидев `Animal<T, S>`, можно никогда не догадаться, что эти типы данных будут указаны в аннотации типа полей `id` и `arial`. В этом случае было бы гораздо предпочтительней дать говорящие имена `Animal<AnimalID, AnimalArial>` или даже `Animal<TAnimalID, TAnimalArial>`, что позволит внутри тела параметризированного типа `Animal` отличать его параметры типа от конкретных объявлений.

Указывается обобщение сразу после идентификатора типа. Это правило остается неизменным даже в тех случаях, когда идентификатор отсутствует (как в случае с безымянным классовым или функциональным выражением), или же и вовсе не предусмотрен (стрелочная функция).

ts

```ts
type Identifier<T> = {};

interface Identifier<T> {}

class Identifier<T> {
    public identifier<T>(): void {}
}

let identifier = class <T> {};

function identifier<T>(): void {}
```

```
let identifier = function <T>(): void {};

let identifier = <T>() => {};
```

Но прежде чем приступить к детальному рассмотрению, нужно уточнить, что правила для функций, функциональных выражений и методов идентичны. Правила для классов ничем не отличаются от правил для классовых выражений. Исходя из этого, все дальнейшие примеры будут приводиться исключительно на классах и функциях.

В случае, когда обобщение указанно псевдониму типа (`type`), область видимости параметров типа ограничена самим выражением.

ts

```
type T1<T> = { f1: T };
```

Область видимости параметров типа при объявлении функции и функционального выражения, включая стрелочное, а также методов, ограничивается их сигнатурой и телом. Другими словами, параметр типа можно использовать в качестве типа при объявлении параметров, возвращаемого значения, а также в допустимых выражениях (аннотация типа, приведение типа и т.д.) расположенных в теле.

ts

```
function f1<T>(p1: T): T {
    let v1: T;

    return v1;
}
```

При объявлении классов (в том числе классовых выражений) и интерфейсов, область видимости параметров типа ограничиваются областью объявления и телом.

ts

```
interface IT1<T> {
    f1: T;
}

class T1<T> {
    public f1: T;
}
```

В случаях, когда класс/интерфейс расширяет другой класс/интерфейс, который объявлен как обобщенный, потомок обязан указать типы для своего предка. Потомок в качестве аргумента типа может указать своему предку не только конкретный тип, но и тип, представляемый собственными параметрами типа.

ts

```ts
interface IT1<T> {}

interface IT3<T> extends IT1<T> {}
interface IT2 extends IT1<string> {}

class T1<T> {}

class T2<T> extends T1<T> implements IT1<T> {}
class T3 extends T1<string> implements IT1<string> {}
```

Если класс/интерфейс объявлен как обобщенный, а внутри него объявлен обобщенный метод, имеющий идентичный параметр типа, то последний в своей области видимости будет перекрывать первый (более конкретно это поведение будет рассмотрено позднее).

ts

```ts
interface IT1<T> {
    m2<T>(p1: T): T;
}

class T1<T> {
    public m1<T>(p1: T): T {
        let v1: T;

        return p1;
    }
}
```

Принадлежность параметра типа к конкретному типу данных устанавливается в момент передачи аргументов типа. При этом конкретные типы данных указываются в паре угловых скобок, а количество конкретных типов должно соответствовать количеству обязательных параметров типа.

ts

```ts
class Animal<T> {
    constructor(readonly id: T) {}
}

var bird: Animal<string> = new Animal('bird'); // Ok
var bird: Animal<string> = new Animal(1); // Error
var fish: Animal<number> = new Animal(1); // Ok
```

Если обобщенный тип указывается в качестве типа данных, то он обязан содержать аннотацию обобщения (исключением является параметры типа по умолчанию, которые рассматриваются далее в главе).

ts

```
class Animal<T> {
    constructor(readonly id: T) {}
}

var bird: Animal = new Animal<string>('bird'); // Error
var bird: Animal<string> = new Animal<string>('bird'); // Ok
```

Когда все обязательные параметры типа используются в параметрах конструктора, при создании экземпляра класса, аннотацию обобщения можно опускать. В таком случае вывод типов определит принадлежность к типам по устанавливаемым значениям. Если параметры являются необязательными и значение не будет передано, то вывод типов определит принадлежность параметров типа к типу данных `unknown`.

ts

```
class Animal<T> {
    constructor(readonly id?: T) {}
}

let bird: Animal<string> = new Animal('bird'); // Ok -> bird:
Animal<string>
let fish = new Animal('fish'); // Ok -> fish: Animal<string>
let insect = new Animal(); // Ok -> insect: Animal<unknown>
```

Относительно обобщенных типов существуют такие понятия, как *открытый* (open) и *закрытый* (closed) тип. Обобщенный тип в момент определения называется *открытым*.

ts

```
class T0<T, U> {} //  T0 - открытый тип
```

Кроме того, обобщенные типы, указанные в аннотации у которых хотя бы один из аргументов типа является параметром типа, также являются открытыми типами.

ts

```
class T1<T> {
  public f: T0<number, T>; // T0 - открытый тип
}
```

И наоборот, если все аргументы типа принадлежат к конкретным типам, то такой обобщенный тип является *закрытым* типом.

ts

```
class T1<T> {
  public f1: T0<number, string>; // T0 - закрытый тип
}
```

Те же самые правила применимы и к функциям, но за одним исключением — вывод типов для примитивных типов определяет принадлежность параметров типа к литеральным типам данных.

ts

```ts
function action<T>(value?: T): T | undefined {
    return value;
}

action<number>(0); // function action<number>(value?: number |
undefined): number | undefined
action(0); // function action<0>(value?: 0 | undefined): 0 | undefined

action<string>('0'); // function action<string>(value?: string |
undefined): string | undefined
action('0'); // function action<"0">(value?: "0" | undefined): "0" |
undefined

action(); // function action<unknown>(value?: unknown): unknown
```

Если параметры типа не участвуют в операциях при создании экземпляра класса и при этом аннотация обобщения не была указана явно, вывод типа теряет возможность установить принадлежность к типу по значению и поэтому устанавливает его принадлежность к типу `unknown`.

ts

```ts
class Animal<T> {
    public name: T;

    constructor(readonly id: string) {}
}

let bird: Animal<string> = new Animal('bird#1');
bird.name = 'bird';
// Ok -> bird: Animal<string>
// Ok -> (property) Animal<string>.name: string

let fish = new Animal<string>('fish#1');
fish.name = 'fish';
// Ok -> fish: Animal<string>
// Ok -> (property) Animal<string>.name: string

let insect = new Animal('insect#1');
insect.name = 'insect';
// Ok -> insect: Animal<unknown>
// Ok -> (property) Animal<unknown>.name: unknown
```

И опять, эти же правила верны и для функций.

ts

```ts
function action<T>(value?: T): T | undefined {
    return value;
}

action<string>('0'); // function action<string>(value?: string |
undefined): string | undefined
action('0'); // function action<"0">(value?: "0" | undefined): "0" |
undefined
action(); // function action<unknown>(value?: unknown): unknown
```

В случаях, когда обобщенный класс содержит обобщенный метод, параметры типа метода будут затенять параметры типа класса.

ts

```ts
type ReturnParam<T, U> = { a: T, b: U };

class GenericClass<T, U> {
    public defaultMethod<T> (a: T, b?: U): ReturnParam<T, U> {
        return { a, b };
    }

    public genericMethod<T> (a: T, b?: U): ReturnParam<T, U> {
        return { a, b };
    }
}

let generic: GenericClass<string, number> = new GenericClass();
generic.defaultMethod('0', 0);
generic.genericMethod<boolean>(true, 0);
generic.genericMethod('0');

// Ok -> generic: GenericClass<string, number>
// Ok -> (method) defaultMethod<string>(a: string, b?: number):
ReturnParam<string, number>
// Ok -> (method) genericMethod<boolean>(a: boolean, b?: number):
ReturnParam<boolean, number>
// Ok -> (method) genericMethod<string>(a: string, b?: number):
ReturnParam<string, number>
```

Стоит заметить, что в *TypeScript* нельзя создавать экземпляры типов представляемых параметрами типа.

ts

```ts
interface CustomConstructor<T> {
    new(): T;
}

class T1<T extends CustomConstructor<T>>{
```

```
    public getInstance(): T {
        return new T(); // Error
    }
}
```

Кроме того, два типа, определяемые классом или функцией, считаются идентичными вне зависимости от того, являются они обобщенными или нет.

ts

```ts
type T1 = {}
type T1<T> = {} // Error -> Duplicate identifier

class T2<T> {}
class T2 {} // Error -> Duplicate identifier

class T3 {
    public m1<T>(): void {}
    public m1(): void {} // Error -> Duplicate method
}

function f1<T>(): void {}
function f1(): void {} // Error -> Duplicate function
```

[34.2] Параметры типа - extends (generic constraints)

Помимо того, что параметры типа можно указывать в качестве конкретного типа, они также могут расширять другие типы, в том числе и другие параметры типа. Такой механизм требуется, когда значения внутри обобщенного типа должны обладать ограниченным набором признаков. Ключевое слово `extends` размещается левее расширяемого типа и правее идентификатора параметра типа `<T extends Type>`. В качестве расширяемого типа может быть указан как конкретный тип данных, так и другой параметр типа. При чем, если один параметр типа расширяет другой, нет разницы в каком порядке они объявляются. Если параметр типа ограничен другим параметром типа, то такое ограничение называют *неприкрытым ограничением типа* (*naked type constraint*),

ts

```ts
class T1 <T extends number> {}
class T2 <T extends number, U extends T> {} // неприкрытое ограничение
```

```
типа
class T3 <U extends T, T extends number> {}
```

Механизм расширения требуется в тех случаях, в которых параметр типа должен обладать заданными характеристиками, необходимыми для выполнения конкретных операций над этим типом.

Для примера рассмотрим случай, когда в коллекции `T` (`Collection<T>`) объявлен метод получения элемента по имени (`getItemByName`).

ts

```ts
class Collection<T> {
    private itemAll: T[] = [];

    public add(item: T): void {
        this.itemAll.push(item);
    }

    public getItemByName(name: string): T {
        return this.itemAll.find(item => item.name === name); // Error -
> Property 'name' does not exist on type 'T'
    }
}
```

При операции поиска в массиве возникнет ошибка. Это происходит по причине того, что в типе `T` не описано свойство `name` . Для того, что бы ошибка исчезла, тип `T` должен расширить тип, в котором описано необходимое свойство `name` . В таком случае предпочтительней будет вариант объявления интерфейса `IName` с последующим его расширением.

ts

```ts
interface IName {
    name: string;
}

class Collection<T extends IName> {
    private itemAll: T[] = [];

    public add(item: T): void {
        this.itemAll.push(item);
    }

    public getItemByName(name: string): T {
        return this.itemAll.find(item => item.name === name); // Ok
    }
}

abstract class Animal {
    constructor(readonly name: string) {}
```

```
}

class Bird extends Animal {}
class Fish extends Animal {}

let birdCollection: Collection<Bird> = new Collection();
birdCollection.add(new Bird('raven'));
birdCollection.add(new Bird('owl'));

let raven: Bird = birdCollection.getItemByName('raven'); // Ok

let fishCollection: Collection<Fish> = new Collection();
fishCollection.add(new Fish('shark'));
fishCollection.add(new Fish('barracuda'));

let shark: Fish = fishCollection.getItemByName('shark'); // Ok
```

Пример, когда параметр типа расширяет другой параметр типа, будет рассмотрен немного позднее.

Также не лишним будет заметить, что когда параметр типа расширяет другой тип, в качестве аргумента типа можно будет передать только совместимый с ним тип.

ts

```
interface Bird { fly(): void; }
interface Fish { swim(): void; }

interface IEgg<T extends Bird> { child: T; }

let v1: IEgg<Bird>; // Ok
let v2: IEgg<Fish>; // Error -> Type 'Fish' does not satisfy the
constraint 'Bird'
```

Кроме того, расширять можно любые подходящие для этого типы, полученные любым доступным путем.

ts

```
interface IAnimal {
    name: string;
    age: number;
}

let animal: IAnimal;

class Bird<T extends typeof animal> {} // T extends IAnimal
class Fish<K extends keyof IAnimal> {} // K extends "name" | "age"
class Insect<V extends IAnimal[K], K extends keyof IAnimal> {} // V
extends string | number
class Reptile<T extends number | string, U extends number & string> {}
```

Помимо прочего, одна важная и не очевидная особенность связана с параметром типа расширяющего `any`. Может показаться, что в таком случае над параметром типа будет возможно производить любые операции допускаемые типом `any`. Но в реальности это не так. Поскольку `any` предполагает выполнение над собой любых операций, то для повышения типобезопасности подобное поведение для типов, представляемых параметрами типа, было отменено.

ts

```
class ClassType<T extends any> {
     private f0: any = {}; // Ok
     private field: T = {}; // Error [0]

     constructor(){
         this.f0.notExistsMethod(); // Ok [1]
         this.field.notExistsMethod(); // Error [2]
     }
 }

 /**
  * Поскольку параметр типа, расширяющий тип any,
  * подрывает типобезопасность программы, то вывод
  * типов такой параметр расценивает как принадлежащий
  * к типу unknown, запрещающий любые операции над собой.
  *
  * [0] тип unknown не совместим с объектным типом {}.
  * [1] Ok на этапе компиляции и Error вовремя выполнения.
  * [2] тип unknown не описывает метода notExistsMethod().
  */
```

[34.3] Параметра типа - значение по умолчанию = (generic parameter defaults)

Помимо прочего, *TypeScript* позволяет указывать для параметров типа значение по умолчанию.

Значение по умолчанию указывается с помощью оператора равно `=`, слева от которого располагается параметр типа, а справа конкретный тип, либо другой параметр типа `T = Type`. Параметры, которым заданы значения по умолчанию, являются необязательными параметрами. Необязательные параметры типа должны быть перечислены строго после обязательных. Если параметр типа указывается в качестве типа по умолчанию, то ему самому должно быть задано значение по умолчанию, либо он должен расширять другой тип.

ts

```ts
class T1<T = string> {} // Ok
class T2<T = U, U> {} // Error -> необязательное перед обязательным
class T3<T = U, U  = number> {} // Ok

class T4<T = U, U extends number> {} // Error -> необязательное перед
обязательным
class T5<U extends number, T = U> {} // Ok.
```

Кроме того, можно совмещать механизм установки значения по умолчанию и механизм расширения типа. В этом случае оператор равно `=` указывается после расширяемого типа.

ts

```ts
class T1 <T extends T2 = T3> {}
```

В момент, когда тип `T` расширяет другой тип, он получает признаки этого типа. Именно поэтому для параметра типа, расширяющего другой тип, в качестве типа по умолчанию можно указывать только совместимый с ним тип.

Чтобы было проще понять, нужно представить два класса, один из которых расширяет другой. В этом случае переменной с типом суперкласса можно в качестве значения присвоить объект его подкласса, но — не наоборот.

ts

```ts
class Animal {
    public name: string;
}

class Bird extends Animal {
    public fly(): void {}
}

let bird: Animal = new Bird(); // Ok
let animal: Bird = new Animal(); // Error
```

Тот же самый механизм используется для параметров типа.

ts

```ts
class Animal {
    public name: string;
}

class Bird extends Animal {
    public fly(): void {}
}
```

```
class T1 <T extends Animal = Bird> {} // Ok
// -------(   Animal   ) = Bird

class T2 <T extends Bird = Animal> {} // Error
// -------(   Bird   ) = Animal
```

Необходимо сделать акцент на том, что вывод типов обращает внимание на необязательные параметры типа только при работе с аргументами этого обобщенного типа. Чтобы было более понятно, вспомним ещё раз, что механизм ограничения параметров типа производится с помощью ключевого слова `extends`. Признаки типа расположенного правее ключевого слова `extends` рассматриваются не только при сопоставлении аргументов типа, но и при выполнении операций над типом, представленным параметром типа. Простыми словами, вывод типов берет во внимание расширенный тип как снаружи (аргумент типа), так и внутри (параметр типа) обобщенного типа.

ts

```
/**
 * T расширяет string...
 */
class A<T extends string> {
    constructor(value?: T) {
        /**
         * ..., что заставляет вывод типов рассматривать
         * значение, принадлежащее к нему, в качестве строкового
         * как внутри...
         */

        if (value) {
            value.charAt(0); // Ok -> ведь value наделено признаками
присущими типу string
        }
    }
}

// ...так и снаружи
let a0 = new A(); // Ok -> let a0: A<string>. string, потому, что
параметр типа ограничен им
let a1 = new A(`ts`); // Ok -> let a1: A<"ts">. literal string, потому,
что он совместим со стринг, но более конкретен
let a2 = new A(0); // Error -> потому, что number не совместим с
ограничивающим аргумент типа типом string
```

Тип, который указывается параметру типа в качестве типа по умолчанию, вообще ничего не ограничивает.

ts

```
// тип string устанавливается типу T в качестве типа по умолчанию...
class B<T = string> {
    constructor(value?: T) {
        if (value) {
            // ..., что не оказывает никакого ограничения ни внутри...
            value.charAt(0); // Error -> тип T не имеет определение
метода charAt
        }
    }
}


// ...ни снаружи
let b0 = new B(); // Ok -> let b0: B<string>
let b1 = new B(`ts`); // Ok -> let b1: B<string>
let b2 = new B(0); // Ok -> let b2: B<number>
```

При отсутствии аргументов типа был бы выведен тип `unknown`, а не тип указанный по умолчанию.

ts

```
// с типом по умолчанию
class B<T = string> {
    constructor(value?: T) {
    }
}

// без типа по умолчанию
class C<T> {
    constructor(value?: T) {
    }
}


let b = new B(); // Ok -> let b: B<string>
let c = new C(); // Ok -> let c: C<unknown>
```

Не будет лишним также рассмотреть отличия этих двух механизмов при работе вывода типов.

ts

```
// ограничение типа T типом string
declare class A<T extends string> {
    constructor(value?: T)
}

// тип string устанавливается типу T в качестве типа по умолчанию
declare class B<T = string> {
    constructor(value?: T)
```

```
}


let a0 = new A(); // Ok -> let a0: A<string>
let b0 = new B(); // Ok -> let b0: B<string>

let a1 = new A(`ts`); // Ok -> let a1: A<"ts">
let b1 = new B(`ts`); // Ok -> let b1: B<string>

let a2 = new A<string>(`ts`); // Ok -> let a2: A<string>
let b2 = new B<string>(`ts`); // Ok -> let b2: B<string>

let a3 = new A<number>(0); // Error
let b3 = new B<number>(0); // Ok -> let b3: B<number>
```

[34.4] Параметры типа - как тип данных

Параметры типа, указанные в угловых скобках при объявлении обобщенного типа, изначально не принадлежат ни к одному типу. Несмотря на это, компилятор расценивает параметры типа как совместимые с такими типами как `any` и `never`, а также самим собой.

ts

```
function f0<T>(p: any): T { // Ok, any совместим с T
    return p;
}

function f1<T>(p: never): T { // Ok, never совместим с T
    return p;
}

function f2<T>(p: T): T { // Ok, T совместим с T
    return p;
}
```

Если обобщенная коллекция в качестве аргумента типа получает тип объединение (`Union`), то все её элементы будут принадлежать к типу объединения. Простыми словами, элемент из такой коллекции не будет, без явного преобразования, совместим ни с одним из вариантов, составляющих тип объединение.

ts

```ts
interface IName { name: string; }

interface IAnimal extends IName {}

abstract class Animal implements IAnimal {
  constructor(readonly name: string) {}
}

class Bird extends Animal {
    public fly(): void {}
}

class Fish extends Animal {
    public swim(): void {}
}

class Collection<T extends IName> {
    private itemAll: T[] = [];

    public add(item: T): void {
        this.itemAll.push(item);
    }

    public getItemByName(name: string): T {
        return this.itemAll.find(item => item.name === name); // Ok
    }
}

let collection: Collection<Bird | Fish> = new Collection();
  collection.add(new Bird('bird'));
  collection.add(new Fish('fish'));

var bird: Bird = collection.getItemByName('bird'); // Error -> Type
'Bird | Fish' is not assignable to type 'Bird'
var bird: Bird = collection.getItemByName('bird') as Bird; // Ok
```

Но операцию приведения типов можно поместить (сокрыть) прямо в метод самой коллекции и тем самым упростить её использование. Для этого метод должен быть обобщенным, а его параметр типа, указанный в качестве возвращаемого из функции, расширять параметр типа самой коллекции.

ts

```ts
// ...

class Collection<T extends IName> {
    private itemAll: T[] = [];

    public add(item: T): void {
        this.itemAll.push(item);
    }
```

```
    // 1. параметр типа U должен расширять параметр типа T
    // 2. возвращаемый тип указан как U
    // 3. возвращаемое значение нуждается в явном преобразовании к типу
U
    public getItemByName<U extends T>(name: string): U {
        return this.itemAll.find(item => item.name === name) as U; // Ok
    }
}

let collection: Collection<Bird | Fish> = new Collection();
 collection.add(new Bird('bird'));
 collection.add(new Fish('fish'));

var bird: Bird = collection.getItemByName('bird'); // Ok
var birdOrFish = collection.getItemByName('bird'); // Bad, var
birdOrFish: Bird | Fish
var bird = collection.getItemByName<Bird>('bird'); // Ok, var bird: Bird
```

Сокрытие приведения типов прямо в методе коллекции повысило “привлекательность” кода. Но, все же, в случаях, когда элемент коллекции присваивается конструкции без явной аннотации типа, появляется потребность вызывать обобщенный метод с аргументами типа.

Кроме того, нужно не забывать, что два разных объявления параметров типа несовместимы, даже если у них идентичные идентификаторы.

ts

```
class Identifier<T> {
    array: T[] = [];

    method<T>(param: T): void  {
        this.array.push(param); // Error, T объявленный в сигнатуре
функции не совместим с типом T объявленном в сигнатуре класса
    }
}
```

Глава 35

Дискриминантное объединение (Discriminated Union)

Чтобы вывод типов мог отличить один тип, входящий в множество `union`, от другого, необходимо, что бы каждый из них определял специальное поле, способное идентифицировать его уникальным образом. Данная глава расскажет как определить подобные поля и научит как на их основе привязывать тип к лексическому окружению.

[35.0] Дискриминантное объединение

Тип `Discriminated Unions` (*дискриминантное объединение*), часто обозначаемое как `Tagged Union` (*размеченное объединение*), так же как и тип `union` (объединение), является множеством типов, перечисленных через прямую черту `|`. Значение, ограниченное дискриминантным объединением, может принадлежать только к одному типу из множества.

ts

```
let v1: T1 | T2 | T3;
```

Из-за того, что все описанное ранее для типа `union` (глава "Типы - Union, Intersection") идентично и для `Tagged Union`, будет более разумно не повторяться, а сделать упор на различия. Но, так как полностью открыть завесу тайны `Tagged Union` на данный

момент будет преждевременным, остается лишь описать детали, к которым рекомендуется вернуться при необходимости.

Несмотря на то, что `Discriminated Union` в большей степени идентичен типу `Union`, все же существует два отличия. Первое отличие заключается в том, что типу `Discriminated Union` могут принадлежать только ссылочные типы данных. Второе отличие в том, что каждому объектному типу (также называемые *варианты*), составляющему `Discriminated Union`, указывается идентификатор варианта *дискриминант*.

Помните, что вывод типов, без помощи разработчика, способен работать лишь с общими для всех типов признаками?

ts

```ts
class Bird {
    fly(): void {}

    toString(): string {
        return 'bird';
    }
}

class Fish {
    swim(): void {}

    toString(): string {
        return 'fish';
    }
}
class Insect {
    crawl(): void {}

    toString(): string {
        return 'insect';
    }
}

function move(animal: Bird | Fish | Insect): void {
    animal.fly(); // Error -> [*]
    animal.swim(); // Error -> [*]
    animal.crawl(); // Error -> [*]

    animal.toString(); // Ok -> [*]

    /**
     * [*]
     *
     * Поскольку вывод типов не может
     * определить к какому конкретно
     * из трех типов принадлежит параметр
     * animal, он не позволяет обращаться к
     * уникальным для каждого типа членам,
```

```
     * коими являются методы fly, swim, crawl.
     *
     * В отличии от этих методов, метод toString
     * определен в каждом из возможных типов,
     * поэтому при его вызове ошибки не возникает.
     */

}
```

Чтобы компилятор мог работать с членами присущих конкретным типам, составляющих дискриминантное объединение, одним из способов является сужение диапазона типов при помощи *дискриминанта*.

ts

```ts
class Bird {
    type: 'bird' = 'bird'; // дискриминант

    fly(): void {}

    toString(): string {
        return 'bird';
    }
}

class Fish {
    type: 'fish' = 'fish'; // дискриминант

    swim(): void {}

    toString(): string {
        return 'fish';
    }
}

class Insect {
    type: 'insect' = 'insect'; // дискриминант

    crawl(): void {}

    toString(): string {
        return 'insect';
    }
}

function move(animal: Bird | Fish | Insect): void {
    if (animal.type === 'bird') {
        animal.fly(); // Ok
    } else if (animal.type === 'fish') {
        animal.swim(); // Ok
    } else {
        animal.crawl(); // Ok
```

```
    }

    animal.toString(); // Ok
}
```

Механизм, с помощью которого разработчик помогает выводу типов, называется *защитники типа* (`type guard`), и будет рассмотрен позднее в одноимённой главе (глава “Типизация - Защитники типа”). А пока стоит сосредоточиться на самих идентификаторах вариантов.

Прежде всего стоит прояснить, что дискриминант, это поле, которое обязательно должно принадлежать к литеральному типу отличному от `unique symbol` и определенное в каждом типе, составляющем *дискриминантное объединение*. Кроме того, поля обязательно должны быть инициализированы при объявлении или в конструкторе. Также не будет лишним напомнить, что список литеральных типов, к которому может принадлежать дискриминант, состоит из `Literal Number` , `Literal String` , `Template Literal String` , `Literal Boolean` и `Literal Enum` .

ts

```
class Bird {
    type: 'bird' = 'bird'; // инициализация в момент объявления поля

    fly(): void {}
}

class Fish {
    type: 'fish' = 'fish'; // инициализация в момент объявления поля

    swim(): void {}
}

class Insect {
    type: 'insect';

    constructor(){
        this.type = 'insect'; // инициализация в конструкторе
    }

    crawl(): void {}
}

function move(animal: Bird | Fish | Insect): void {}
```

В случае, когда типы полей являются уникальными для всего множества, они идентифицируют только свой тип.

ts

```
class Bird {
    groupID: 0 = 0;
```

```
    fly(): void {}
}

class Fish {
    groupID: 1 = 1;

    swim(): void {}
}

class Insect {
    groupID: 2 = 2;

    crawl(): void {}
}

// groupID 0 === Bird
// groupID 1 === Fish
// groupID 2 === Insect
```

Тогда, когда тип поля не является уникальным, он идентифицирует множество типов, у которых совпадают типы одноимённых идентификаторов вариантов.

ts

```
class Bird {
    groupID: 0 = 0;

    fly(): void {}
}

class Fish {
    groupID: 0 = 0;

    swim(): void {}
}

class Insect {
    groupID: 1 = 1;

    crawl(): void {}
}

// groupID 0 === Bird | Fish
// groupID 1 === Insect
```

Количество полей, которые служат идентификаторами вариантов, может быть любым.

ts

```
enum AnimalTypes {
    Bird = 'bird',
```

```
    Fish = 'fish',
}


class Bird {
    type: AnimalTypes.Bird = AnimalTypes.Bird;

    fly(): void {}
}
class Robin extends Bird {
    id: 0 = 0;
}
class Starling extends Bird {
    id: 1 = 1;
}


class Fish {
    type: AnimalTypes.Fish = AnimalTypes.Fish;

    swim(): void {}
}
class Shark extends Fish {
    id: 0 = 0;
}
class Barracuda extends Fish {
    id: 1 = 1;
}



declare const animal: Robin | Starling | Shark | Barracuda;



if (animal.type === AnimalTypes.Bird) {
    /**
     * В области видимости этого блока if
     * константа animal принадлежит к типу Bird или Starling
     */
    animal; // const animal: Robin | Starling

    if (animal.id === 0) {
        /**
         * В области видимости этого блока if
         * константа animal принадлежит к типу Robin
         */
        animal; // const animal: Robin
    } else {
        /**
         * В области видимости этого блока else
         * константа animal принадлежит к типу Starling
         */
```

```
        animal; // const animal: Starling
    }
} else {
    /**
     * В области видимости этого блока if
     * константа animal принадлежит к типу Shark или Barracuda
     */

    animal; // const animal: Shark | Barracuda

    if (animal.id === 0) {
        /**
         * В области видимости этого блока if
         * константа animal принадлежит к типу Shark
         */
        animal; // const animal: Shark
    } else {
        /**
         * В области видимости этого блока else
         * константа animal принадлежит к типу Barracuda
         */

        animal; // const animal: Barracuda
    }
}
```

При необходимости декларирования поля, выступающего в роли дискриминанта, в интерфейсе, ему указывается более общий совместимый тип. Для литерального строкового типа, это тип `string`, для литерального числового, это `number` и т.д.

ts

```
interface IT {
  /**
   * Дискриминантное поле
   */
  type: string; // это поле предполагается использовать в качестве
дискриминанта поля
}


class A implements IT {
  type: "a" = "a"; // переопределение более конкретным типом
}
class B implements IT {
  type: "b" = "b"; // переопределение более конкретным типом
}


function valid(value: A | B) {
```

```
  if (value.type === "a") {
    // здесь value расценивается как принадлежащее к типу A
  } else if (value.type === "b") {
    // здесь value расценивается как принадлежащее к типу B
  }

  // здесь value расценивается как тип обладающий общими для A b B
признаками
}
```

Глава 36

Импорт и экспорт только типа

На самом деле привычный всем механизм импорта\экспорта таит в себе множество курьезов способных нарушить ожидаемый ход выполнения программы. Помимо детального рассмотрения каждого из них, текущая глава также расскажет о способах их разрешения.

[36.0] Предыстория возникновения import type и export type

Представьте сценарий, по которому существуют два модуля, включающих экспорт класса. Один из этих классов использует другой в аннотации типа своего единственного параметра конструктора, что требует от модуля в котором он реализован, импорта другого модуля. Подвох заключается в том, что несмотря на использование класса в качестве типа, модуль в котором он определен вместе с его содержимым, все равно будет скомпилирован в конечную сборку.

ts

```ts
// @filename: ./SecondLevel.ts
export class SecondLevel {

}
```

ts

```ts
// @filename: ./FirstLevel.ts
import {SecondLevel} from "./SecondLevel";

export class FirstLevel {
    /**
     * класс SecondLevel используется
     * только как тип
     */
    constructor(secondLevel: SecondLevel){}
}
```

ts

```ts
// @filename: ./index.js
export { FirstLevel } from "./FirstLevel";
```

js

```js
// @info: скомпилированный проект

// @filename: ./SecondLevel.js
export class SecondLevel {
}

// @filename: ./FirstLevel.js
/**
 * Несмотря на то, что от класса SecondLevel не осталось и следа,
 * модуль *, в котором он определен, все равно включен в сборку.
 */
import "./SecondLevel"; // <-- *
export class FirstLevel {
    /**
     * класс SecondLevel используется
     * только как тип
     */
    constructor(secondLevel) { }
}

// @filename: ./index.js
export { FirstLevel } from "./FirstLevel";
```

При использовании допустимых *JavaScript* конструкций исключительно в качестве типа, было бы разумно ожидать, что конечная сборка не будет обременена модулями, в которых они определены. Кроме того, конструкции, присущие только *TypeScript*, не попадают в конечную сборку, в отличие от модулей, в которых они определенны. Если в нашем примере поменять тип конструкции `SecondLevel` с класса на интерфейс, то модуль `./FirstLevel.js` все равно будет содержать импорт модуля `./SecondLevel.js`, содержащего экспорт пустого объекта `export {};`. Не лишним будут обратить внимание, что в случае с интерфейсом, определяющий его модуль мог содержать и другие конструкции. И если бы среди этих конструкций оказались

допустимые с точки зрения *JavaScript*, то они, на основании изложенного ранее, попали бы в конечную сборку. Даже если бы вообще не использовались.

Это поведение привело к тому, что в *TypeScript* появился механизм импорта и экспорта только типа. Этот механизм позволяет устранить рассмотренные случаи. Тем не менее, имеется несколько нюансов, которые будут подробно изложены далее.

[36.1] import type и export type - форма объявления

Форма уточняющего *импорта и экспорта только типа* включает в себя ключевое слово `type`, идущее следом за ключевым словом `import` либо `export`.

ts
```ts
import type { Type } from './type';
export type { Type };
```

Ключевое слово `type` можно размещать в выражениях импорта, экспорта, а также ре-экспорта.

ts
```ts
// @filename: ./ClassType.ts

export class ClassType {

}
```

ts
```ts
// @filename: ./index.js

import type { ClassType } from "./types"; // Ok -> импорт только типа

export type { ClassType }; // Ok -> экспорт только типа
```

ts
```ts
// @filename: ./index.js

export type { ClassType } from "./types"; // Ok -> ре-экспорт только
типа
```

Единственное отличие импорта и экспорта только типа от обычных одноименных инструкций состоит в невозможности импортировать в одной форме обычный импорт/экспорт и по умолчанию.

ts

```ts
// @filename: ./types.ts

export default class DefaultClassType {}
export class ClassType {}
```

ts

```ts
// @filename: ./index.js

// пример с обычным импортом

import DefaultClassType, { ClassType } from "./types"; // Ok -> обычный импорт
```

ts

```ts
// @filename: ./index.js

// неправильный пример с импортом только типа

import type DefaultClassType, { ClassType } from "./types"; // Error -> импорт только типа

/**
 * [0] A type-only import can specify a default import or named bindings, but not both.
 */
```

Как можно почерпнуть из текста ошибки, решение заключается в создании отдельных форм импорта.

ts

```ts
// @filename: ./index.js

// правильный пример с импортом только типа

import type DefaultClassType from "./types"; // Ok -> импорт только типа по умолчанию
import type { ClassType } from "./types"; // Ok -> импорт только типа
```

[36.2] Импорт и экспорт только типа на практике

Прежде всего перед рассматриваемым механизмом стоит задача недопущения использования импортированных или экспортированных только как тип конструкций и значений в роли, отличной от обычного типа. Другими словами, допустимые *JavaScript* конструкции и значения нельзя использовать в привычных для них выражениях. Нельзя создавать экземпляры классов, вызывать функции и использовать значения, ассоциированные с переменными.

ts

```ts
// filename: ./types.ts

export class ClassType {}
export interface IInterfaceType {}
export type AliasType = {};

export const o = {person: '🧟'};

export const fe = () => {};
export function fd() {}
```

ts

```ts
import type { o, fe, fd, ClassType, IInterfaceType } from './types'; // Ok

/**
 * * - '{{NAME}}' cannot be used as a value because it was imported using 'import type'.
 */

let person = o.person; // Error -> *
fe(); // Error -> *
fd(); // Error -> *
new ClassType(); // Error -> *
```

Не сложно догадаться, что значения и функции импортированные или экспортированные только как типы необходимо использовать в совокупности с другим механизмом, называемым *запрос типа*.

ts

```
import type { o, fe, fd, ClassType, IInterfaceType } from './types';

/**
 * v2, v3 и v4 используют механизм
 * запроса типа
 */

let v0: IInterfaceType; // Ok -> let v0: IInterfaceType
let v1: ClassType; // Ok -> let v1: ClassType
let v2: typeof fd; // Ok -> let v2: () => void
let v3: typeof fe; // Ok -> let v3: () => void
let v4: typeof o; // Ok -> let v4: {person: string;}
```

Будет не лишним уточнить, что классы, экспортированные как уточнённые, не могут участвовать в механизме наследования.

ts

```
// @filename: Base.ts

export class Base {}
```

ts

```
// @filename: index.js

import type { Base } from "./Base";

/**
 * Error -> 'Base' cannot be used as a value because it was imported
using 'import type'.
 */
class Derived extends Base {}
```

[36.3] Вспомогательный флаг --importsNotUsedAsValues

Другая задача, решаемая с помощью данного механизма, заключается в управлении включения модулей в конечную сборку при помощи вспомогательного флага `--importsNotUsedAsValues`, значение которого может принадлежать к одному из трех вариантов. Но прежде чем познакомится с каждым из них, необходимо поглубже погрузиться в природу возникновения необходимости в данном механизме.

Большинство разработчиков, пользуясь механизмом импорта/экспорта в повседневной работе, даже не подозревают, что с ним связанно немало различных трудностей, возникающих из-за механизмов, призванных оптимизировать код. Но для начала рассмотрим несколько простых вводных примеров.

Представьте ситуацию, при которой один модуль импортирует необходимый ему тип, представленный интерфейсом.

ts

```ts
// @filename IPerson.ts

export interface IPerson {
    name: string;
}
```

ts

```ts
// @filename action.ts

import { IPerson } from './IPerson';

function action(person: IPerson) {
    // ...
}
```

Поскольку интерфейс является конструкцией присущей исключительно *TypeScript*, то неудивительно, что после компиляции от неё и модуля, в которой она определена, не останется и следа.

js

```js
// после компиляции @file action.js

function action(person) {
    // ...
}
```

Теперь представьте, что один модуль импортирует конструкцию, представленную классом, который задействован в логике уже знакомой нам функции `action()`.

ts

```ts
// @file IPerson.ts

export interface IPerson {
    name: string;
}
```

ts

```ts
// @file IPerson.ts

export class Person {
    constructor(readonly name: string) {}

    toString() {
        return `[person ${this.name}]`;
    }
}
```

ts

```ts
// @file Person.ts

import { IPerson } from './IPerson';
import { Person } from './Person';

function action(person: IPerson) {
    new Person(person);
}
```

js

```ts
// после компиляции @file action.js

import { Person } from './Person';


function action(person) {
    new Person(person);
}
```

В этом случае класс `Person` был включён в скомпилированный файл, поскольку необходим для правильного выполнения программы.

А теперь представьте ситуацию, когда класс `Person` задействован в том же модуле `action.ts`, но исключительно в качестве типа. Другими словами, он не задействован в логике работы модуля.

ts

```ts
// @file Person.ts

export class Person {
    constructor(readonly name:string) {}

    toString() {
        return `[person ${this.name}]`;
    }
}
```

ts

```ts
// @file action.ts

import { Person } from './Person';

function action(person: Person) {
    //...
}
```

Подумайте, что должна включать в себя итоговая сборка? Если вы выбрали вариант идентичный первому, то вы совершенно правы! Поскольку класс `Person` используется в качестве типа, то нет смысла включать его в результирующий файл.

js

```ts
// после компиляции @file action.js

function action(person) {
    //...
}
```

Подобное поведение кажется логичным и возможно благодаря механизму, называемому *import elision*. Этот механизм определяет, что конструкции, которые теоретически могут быть включены в скомпилированный модуль, требуются ему исключительно в качестве типа. И, как уже можно было догадаться, именно с этим механизмом и связаны моменты, мешающие оптимизации кода.

Рассмотрим пример состоящий из двух модулей. Первый модуль экспортирует объявленные в нем интерфейс и функцию, использующую этот интерфейс в аннотации типа своего единственного параметра. Второй модуль лишь ре-экспортирует интерфейс и функцию из первого модуля.

ts

```ts
// @filename module.ts

export interface IActionParams {}
export function action(params: IActionParams) {}
```

ts

```ts
// @filename re-export.ts

import { IActionParams, action } from './types';

/**
 * Error -> Re-exporting a type when the '--isolatedModules' flag is
provided requires using 'export type'
 */
export { IActionParams, action };
```

Поскольку компиляторы, как TypeScript, так и Babel, неспособны определить, является ли конструкция `IActionParams` допустимой для *JavaScript* в контексте файла, существует вероятность возникновения ошибки. Простыми словами, механизмы обоих компиляторов не знают нужно ли удалять следы, связанные с `IActionParams` из скомпилированного *JavaScript* кода или нет. Именно поэтому существует флаг `--isolatedModules`, активация которого заставляет компилятор предупреждать об опасности данной ситуации.

Механизм уточнения способен разрешить возникающие перед *import elision* трудности ре-экспорта модулей, предотвращению которых способствует активация флага `--isolatedModules`.

Рассмотренный выше случай можно разрешить с помощью явного уточнения формы импорта/экспорта.

ts

```ts
// @filename: re-export.ts

import { IActionParams, action } from './module';

/**
 * Явно указываем, что IActionParams это тип.
 */
export type { IActionParams };
export { action };
```

Специально введенный и ранее упомянутый флаг `--importsNotUsedAsValues` ожидает одно из трех возможных на данный момент значений - `remove`, `preserve` или `error`.

Значение `remove` реализует поведение по умолчанию и которое обсуждалось на протяжении всей главы.

Значение `preserve` способно разрешить проблему, возникающую при экспорте так называемых *сайд-эффектов*.

ts

```ts
// @filename: module-with-side-effects.ts

function incrementVisitCounterLocalStorage() {
    // увеличиваем счетчик посещаемости в localStorage
}

export interface IDataFromModuleWithSideEffects {};

incrementVisitCounterLocalStorage(); // ожидается, что вызов произойдет
в момент подключения модуля
```

ts

```ts
// @filename: index.js

import { IDataFromModuleWithSideEffects } from './module';

let data: IDataFromModuleWithSideEffects = {};
```

Несмотря на то, что модуль `module-with-side-effects.ts` задействован в коде, его содержимое не будет включено в скомпилированную программу, поскольку компилятор исключает импорты конструкций, не участвующих в её логике. Таким образом функция `incrementVisitCounterLocalStorage()` никогда не будет вызвана, а значит программа не будет работать корректно!

js

```js
// @filename: index.js
// после компиляции

let data = {};
```

Решение этой проблемы заключается в повторном указании импорта всего модуля. Но не всем такое решение кажется очевидным.

ts

```ts
import {IDataFromModuleWithSideEffects} from './module-with-side-
effects';
import './module-with-side-effects'; // импорт всего модуля

let data: IDataFromModuleWithSideEffects = {};
```

Теперь программа выполнится так как и ожидалось. То есть модуль `module-with-side-effects.ts` включен в её состав.

js

```js
// @filename: index.js
// после компиляции
```

```
import './module-with-side-effects.js';

let data = {};
```

Кроме того, сама *ide* укажет на возможность уточнения *импорта только типов*, что в свою очередь должно подтолкнуть на размышление об удалении импорта при компиляции.

ts

```
import { IDataFromModuleWithSideEffects } from './module-with-side-
effects'; // This import may be converted to a type-only import.ts(1372)
```

Также флаг `preserve` в отсутствие уточнения поможет избавиться от повторного указания импорта. Простыми словами значение `preserve` указывает компилятору импортировать все модули полностью.

ts

```
// @filename: module-with-side-effects.ts

function incrementVisitCounterLocalStorage() {
    // увеличиваем счетчик посещаемости в localStorage
}

export interface IDataFromModuleWithSideEffects {};

incrementVisitCounterLocalStorage();
```

ts

```
// @filename: module-without-side-effects.ts

export interface IDataFromModuleWithoutSideEffects {};
```

ts

```
// @filename: index.js

// Без уточнения
import { IDataFromModuleWithSideEffects } from './module-with-side-
effects';
import { IDataFromModuleWithoutSideEffects } from './module-without-
side-effects';

let dataFromModuleWithSideEffects: IDataFromModuleWithSideEffects = {};
let dataFromModuleWithoutSideEffects: IDataFromModuleWithoutSideEffects
= {};
```

Несмотря на то, что импортировались исключительно конструкции-типы, как и предполагалось, модули были импортированы целиком.

js

```
// после компиляции @file index.js

import './module-with-side-effects';
import './module-without-side-effects';

let dataFromModuleWithSideEffects = {};
let dataFromModuleWithoutSideEffects = {};
```

В случае уточнения поведение при компиляции останется прежним. То есть импорты в скомпилированный файл включены не будут.

ts

```
// @filename: index.js

// С уточнением
import type { IDataFromModuleWithSideEffects } from './module-with-side-
effects';
import type { IDataFromModuleWithoutSideEffects } from './module-
without-side-effects';


let dataFromModuleWithSideEffects: IDataFromModuleWithSideEffects = {};
let dataFromModuleWithoutSideEffects: IDataFromModuleWithoutSideEffects
= {};
```

Импорты модулей будут отсутствовать.

js

```
// @filename: index.js
// после компиляции

let dataFromModuleWithSideEffects = {};
let dataFromModuleWithoutSideEffects = {};
```

Если флаг `--importsNotUsedAsValues` имеет значение `error`, то при импортировании типов без явного уточнения будет считаться ошибочным поведением.

ts

```
// @filename: index.js

/**
 *
 * [0][1] Error > This import is never used as a value and must use
'import type' because the 'importsNotUsedAsValues' is set to
'error'.ts(1371)
 */
```

```
import { IDataFromModuleWithSideEffects } from './module-with-side-
effects';
import { IDataFromModuleWithoutSideEffects } from './module-without-
side-effects';

let dataFromModuleWithSideEffects: IDataFromModuleWithSideEffects = {};
let dataFromModuleWithoutSideEffects: IDataFromModuleWithoutSideEffects
= {};
```

Скомпилированный код после устранения ошибок, то есть после уточнения, включать в себя импорты не будет.

В заключение стоит заметить, что, теоретически, уточнение класса, используемого только в качестве типа, способно ускорить компиляцию, поскольку это избавляет компилятор от ненужных проверок на вовлечении его в логику работы модуля. Кроме того, уточнение формы импорта/экспорта — это ещё один способ сделать код более информативным.

Глава 37

Утверждение типов (Type Assertion)

Получение значения, которое несоответствует ожидаемому типу, является обычным делом для типизированных языков. Понимание причин, лежащих в основе несоответствий, а также всевозможные способы их разрешений, являются целями данной главы.

[37.0] Утверждение типов - общее

При разработке приложений на языках со статической типизацией, время от времени может возникнуть нестыковка из-за несоответствия типов. Простыми словами, приходится работать с объектом, принадлежащим к известному типу, но ограниченному более специализированным (менее конкретным) интерфейсом.

В *TypeScript* большинство операций с несоответствием типов приходится на работу с *dom* (*Document Object Model*).

В качестве примера можно рассмотреть работу с таким часто используемым методом, как `querySelector()` . Но для начала вспомним, что в основе составляющих иерархию dom-дерева объектов лежит базовый тип `Node` , наделенный минимальными признаками, необходимыми для построения коллекции. Базовый тип `Node` , в том числе, расширяет и тип `Element` , который является базовым для всех элементов dom-дерева и обладает знакомыми всем признаками, необходимыми для работы с элементами dom, такими как атрибуты (`attributes`), список классов (`classList`), размеры клиента

(`client*`) и другими. Элементы dom-дерева можно разделить на те, что не отображаются (унаследованные от `Element` , как например `script` , `link`) и те, что отображаются (например `div` , `body`). Последние имеют в своей иерархии наследования тип `HTMLElement` , расширяющий `Element` , который привносит признаки, присущие отображаемым объектам, как например координаты, стили, свойство `dataset` и т.д.

Возвращаясь к методу `querySelector()` , стоит уточнить, что результатом его вызова может стать любой элемент, находящийся в dom-дереве. Если бы в качестве типа возвращаемого значения был указан тип `HTMLElement` , то операция получения элемента `<script>` или `<link>` завершилась бы неудачей, так как они не принадлежат к этому типу. Именно поэтому методу `querySelector()` в качестве типа возвращаемого значения указан более базовый тип `Element` .

ts

```ts
// <canvas id="stage" data-inactive="false"></canvas>

const element: Element = document.querySelector('#stage');
const stage: HTMLElement = element // Error, Element is not assignable
to type HTMLElement
```

Но, при попытке обратится к свойству `dataset` через объект, полученный с помощью `querySelector()` , возникнет ошибка, так как у типа `Element` отсутствует данное свойство. Факт, что разработчику известен тип, к которому принадлежит объект по указанному им селектору, дает ему основания попросить вывод типов пересмотреть свое отношение к типу конкретного объекта.

Попросить - дословно означает, что разработчик может лишь попросить вывод типов пересмотреть отношение к типу. Но решение разрешить операцию или нет все равно остается за последним.

Выражаясь человеческим языком, в *TypeScript* процесс, вынуждающий вывод типов пересмотреть свое отношение к какому-либо типу, называется *утверждением типа* (`Type Assertion`).

Формально утверждение типа похоже на *преобразование* (приведение) типов (*type conversion*, *typecasting*), но, поскольку в скомпилированном коде от типов не остается и следа, то, по факту, это совершенно другой механизм. Именно поэтому он и называется *утверждение*. Утверждая тип, разработчик говорит компилятору — *“поверь мне, я знаю, что делаю”* (*Trust me, I know what I'm doing*).

Нельзя не уточнить, что хотя в *TypeScript* и существует термин утверждение типа, по ходу изложения в качестве синонимов будут употребляться слова преобразование, реже — приведение. А так же, не будет лишним напомнить, что приведение — это процесс в котором объект одного типа преобразуется в объект другого типа.

[37.1] Утверждение типа с помощью <Type> синтаксиса

Одним из способов указать компилятору на принадлежность значения к заданному типу является механизм утверждения типа при помощи угловых скобок `<ConcreteType>`, заключающих в себе конкретный тип, к которому и будет выполняться преобразование. Утверждение типа располагается строго перед выражением, результатом выполнения которого, будет преобразуемый тип.

ts

```
<ToType>FromType
```

Перепишем предыдущий код и исправим в нем ошибку, связанную с несоответствием типов.

ts

```
// <canvas id="stage" data-inactive="false"></canvas>

const element: Element = document.querySelector('#stage');

const stage: HTMLElement = <HTMLElement>element // Ok
stage.dataset.inactive = 'true';
```

Если тип, к которому разработчик просит преобразовать компилятор, не совместим с преобразуемым типом, то в процессе утверждения возникнет ошибка.

ts

```
class Bird {
    public fly(): void {}
}

class Fish {
    public swim(): void {}
}

let bird: Bird = new Bird();
let fish: Fish = <Fish>bird; // Ошибка, 'Bird' не может быть
преобразован в 'Fish'
```

Кроме того, существуют ситуации, в которых возникает необходимость множественного последовательного преобразования. Ярким примером являются значения полученные от *dom* элементов, которые воспринимаются разработчиком как числовые или логические, но по факту принадлежат к строковому типу.

ts

```ts
// <div id="#container"></div>

let element = document.querySelector('#container') as HTMLElement;
let { width, height } = element.style;
let area: number = width * height; // ошибка -> width и height типа
'string'
```

Дело в том, что в *TypeScript* невозможно привести тип `string` к типу `number`.

ts

```ts
// <div id="#container"></div>

let element = document.querySelector('#container') as HTMLElement;
let { width: widthString, height: heightString } = element.style;

let width: number = <number>widthString; // Ошибка -> тип 'string' не
может быть преобразован  в 'number'
let height: number = <number>heightString; // Ошибка -> тип 'string' не
может быть преобразован  в 'number'
```

Но осуществить задуманное можно преобразовав тип `string` сначала в тип `any`, а уже затем — в тип `number`.

ts

```ts
// <div id="#container"></div>

let element = document.querySelector('#container') as HTMLElement;
let { width: widthString, height: heightString } = element.style;

let width: number = <number><any>widthString; // Ok
let height: number = <number><any>heightString; // Ok

let area: number = width * height; // Ok
```

Стоит также заметить, что данный способ утверждения типа, кроме синтаксиса, больше ничем не отличается от указания с помощью оператора `as`.

[37.2] Утверждение типа с помощью оператора as

В отличие от синтаксиса угловых скобок, которые указываются перед преобразуемым типом, оператор `as` указывается между преобразуемым и типом, к которому требуется преобразовать.

ts
```
FromType as ToType
```

Для демонстрации оператора `as` рассмотрим ещё один часто встречающийся случай, требующий утверждения типов.

Обычное дело: при помощи метода `querySelector()` получить объект, принадлежащий к типу `HTMLElement` и подписать его на событие `click`. Задача заключается в том, что при возникновении события, нужно изменить значение поля `dataset`, объявленного в типе `HTMLElement`. Было бы нерационально снова получать ссылку на объект при помощи метода `querySelector()`, ведь нужный объект хранится в свойстве объекта события `target`. Но дело в том, что свойство `target` имеет тип `EventTarget`, который не находится в иерархической зависимости с типом `HTMLElement` имеющим нужное свойство `dataset`.

ts
```
// <span id="counter"></span>

let element = document.querySelector('#counter') as HTMLElement;
element.dataset.count = (0).toString();

element.addEventListener('click', ({ target }) => {
    let count: number = target.dataset.count; // Error -> Property
'dataset' does not exist on type 'EventTarget'
});
```

Но эту проблему легко решить с помощью оператора утверждения типа `as`. Кроме того, с помощью этого же оператора можно привести тип `string`, к которому принадлежат все свойства находящиеся в `dataset`, к типу `any`, а уже затем к типу `number`.

ts
```
let element = document.querySelector('#counter') as HTMLElement;
element.dataset.count = (0).toString();
```

```
element.addEventListener('click', ({ target }) => {
    let element = target as HTMLElement;
    let count: number = element.dataset.count as any as number;

    element.dataset.count = (++count).toString();
});
```

В случае несовместимости типов возникнет ошибка.

ts

```
class Bird {
    public fly(): void {}
}

class Fish {
    public swim(): void {}
}

let bird: Bird = new Bird();
let fish: Fish = bird as Fish; // Ошибка, 'Bird' не может быть
преобразован в 'Fish'
```

Ещё одна острая необходимость, требующая утверждения типа, возникает тогда, когда разработчику приходится работать с объектом, ссылка на который ограничена более общим типом, как например `any` .

Факт, что над значением, принадлежащему к типу `any` , разрешено выполнение любых операций, означает, что компилятор их не проверяет. Другими словами, разработчик, указывая тип `any` , усложняет процесс разработки, мешая компилятору проводить статический анализ кода, а также лишает себя помощи со стороны редактора кода. Когда разработчику известно к какому типу принадлежит значение, можно попросить компилятор изменить мнение о принадлежности значения к его типу с помощью механизма утверждения типов.

ts

```
class DataProvider {
    constructor(readonly data: any) {}
}

let provider: DataProvider = new DataProvider('text');

var charAll: string[] = provider.data.split(''); // Ок
var charAll: string[] = provider.data.sPlIt(''); // Ошибка во время
выполнения программы
var charAll: string[] = (provider.data as string).split(''); // Ок

let dataString: string = provider.data as string;
var charAll: string[] = dataString.split(''); // Ок
```

Напоследок, стоит сказать, что выражения, требующие *утверждения типа*, при работе с *dom api* — это неизбежность. Кроме того, для работы с методом `document.querySelector()`, который был использован в примерах к этой главе, вместо приведения типов с помощью операторов `<Type>` или `as` предпочтительней конкретизировать тип с помощью обобщения, которые рассматриваются в главе “Типы - Обобщения (Generics)”. Но в случае, если утверждение требуется для кода, написанного самим разработчиком, то, скорее всего, это следствие плохо продуманной архитектуры.

[37.3] Приведение (утверждение) к константе (const assertion)

Ни для кого не секрет, что с точки зрения *JavaScript*, а следовательно и *TypeScript*, все примитивные литеральные значения являются константными значениями. С точки зрения среды исполнения два эквивалентных литерала любого литерального типа являются единым значением. То есть, среда исполнения расценивает два строковых литерала `'text'` и `'text'` как один литерал. Тоже справедливо и для остальных литералов, к которым помимо типа `string` также относятся типы `number`, `boolean` и `symbol`.

Тем не менее, сложно найти разработчика *TypeScript*, не испытавшего трудностей, создаваемых выводом типов, при определении конструкций, которым предстоит проверка на принадлежность к литеральному типу.

ts

```
type Status = 200 | 404;
type Request = { status: Status }
```

```
let status = 200;

let request: Request = { status }; // Error, TS2322: Type 'number' is
not assignable to type 'Status'.
```

В коде выше ошибка возникает по причине того, что вывод типов определяет принадлежность значения переменной `status` к типу `number`, а не литеральному числовому типу `200`.

ts

```
// вывод типов видит как
let status: number = 200

// в, то время как требуется так
let port: 200 = 200;
```

Прежде всего не будет лишним упомянуть, что данную проблему можно решить с помощью механизма утверждения при помощи таких операторов как `as` и угловых скобок `<>`.

ts

```
type Status = 200 | 404;
type Request = { status: Status }

let status = 200;

// утверждаем компилятору..
let request: Request = { status: status as 200 }; // ...с помощью as
оператора
// let request: Request = { status: <200>status }; // ...или с помощью
угловых скобок
// ..., что он должен рассматривать значение, ассоциированное с as, как
значение, принадлежащие к литеральному типу '200'
```

Но лучшим решением будет специально созданный для подобных случаев механизм, позволяющий производить утверждение к константе.

Константное утверждение производится с помощью оператора `as` или угловых скобок `<>` и говорит компилятору, что значение является константным.

ts

```
type Status = 200 | 404;
type Request = { status: Status }

let status = 200 as const;
// let status = <const>200;
```

```
let request: Request = { status }; // Ok
```

Утверждение, что значение является константным, заставляет вывод типов расценивать его как принадлежащее к литеральному типу. Утверждение к константе массива заставляет вывод типов определять его принадлежность к типу `readonly tuple`.

ts

```
let a = [200, 404]; // let a: number[]

let b = [200, 404] as const; // let b: readonly [200, 404]
let c = <const>[200, 404]; // let c: readonly [200, 404]
```

В случае с объектным типом, утверждение к константе рекурсивно помечает все его поля как `readonly`. Кроме того, все его поля, принадлежащие к примитивным типам, расцениваются как литеральные типы.

ts

```
type NotConstResponseType = {
    status: number;
    data: {
        role: string;
    };
}

type ConstResponseType = {
    status: 200 | 404;
    data: {
        role: 'user' | 'admin';
    };
}

let a = { status: 200, data: { role: 'user' }}; // NotConstResponseType

let b = { status: 200, data: { role: 'user' }} as const; //
ConstResponseType
let c = <const>{ status: 200, data: { role: 'user' }}; //
ConstResponseType
```

Но стоит помнить, что утверждение к константе применимо исключительно к литералам таких типов, как `number`, `string`, `boolean`, `array` и `object`.

ts

```
let a = 'value' as const; // Ok - 'value' является литералом, let a:
"value"
let b = 100 as const; // Ok - 100 является литералом, let b: 100
let c = true as const; // Ok - true является литералом, let c: true
```

```
let d = [] as const; // Ok - [] является литералом, let d: readonly []
let e = { f: 100 } as const; // Ok - {} является литералом, let e:
{readonly f: 100}

let value = 'value'; // let value: string
let array = [0, 1, 2]; // let array: number[]
let object = { f: 100 }; // let object: {f: number}

let f = value as const; // Ошибка, value — это ссылка на идентификатор,
хранящий литерал
let g = array as const; // Ошибка, array — это ссылка на идентификатор,
хранящий ссылку на массив
let h = object as const; // Ошибка, object — это ссылка на
идентификатор, хранящий ссылку на объект
```

После рассмотрения всех случаев утверждения к константе (примитивных, массивов и объектных типов) может сложиться впечатление, что в *TypeScript*, наконец, появились структуры, которые справедливо было бы назвать *полноценными константами*, неизменяемыми ни при каких условиях. И это, отчасти, действительно так. Но дело в том, что на данный момент, принадлежность объектных и массивоподобных типов к константе зависит от значений, с которыми они ассоциированы.

В случае, когда литералы ссылочных типов (массивы и объекты) ассоциированы со значением также принадлежащим к ссылочному типу, они представляются такими, какими были на момент ассоциации. Кроме того, поведение механизма приведения к константе зависит от другого механизма — деструктуризации.

ts

```
let defaultObject = { f: 100 }; // let defaultObject: {f: number}
let constObject = { f: 100 } as const; // let constObject: {readonly f:
100}

let defaultArray = [0, 1, 2]; // let defaultArray: number[]
let constArray = [0, 1, 2] as const; // let constArray: readonly [0, 1,
2]

// o0 иммутабельный (неизменяемый) объект
let o0 = { f: { f: 100 } } as const; // {readonly f: {readonly f: 100}}
// o1.f имеет модификатор readonly, o1.f.f - мутабельный (изменяемый)
объект
let o1 = { f: defaultObject } as const; // {readonly f: {f: number}}
// o2 иммутабельный (неизменяемый) объект
let o2 = { ...defaultObject } as const; // {readonly f: number}
// o3.f и o3.f.f иммутабельные (неизменяемые) объекты
let o3 = { f: { ...defaultObject } } as const; // {readonly f:
{readonly f: number}}

// o4.f и o4.f.f иммутабельные (неизменяемые) объекты
let o4 = { f: constObject } as const; // let o4: {readonly f: {readonly
f: 100}}
// o5 иммутабельный (неизменяемый) объект
```

```
let o5 = { ...constObject } as const; // let o5: {readonly f: 100}
// об иммутабельный (неизменяемый) объект
let o6 = { f: { ...constObject } } as const; // {readonly f: {readonly
f: 100}}
```

По причине, что объектные типы данных, хранящиеся в массиве, подчиняются описанным выше правилам, подробное рассмотрение процесса утверждения массива к константе будет опущено.

И последнее, о чем стоит упомянуть — утверждение к константе применимо только к простым выражениям.

ts

```
let a = (Math.round(Math.random() * 1) ? 'yes' : 'no') as const; //
Ошибка
let b = Math.round(Math.random() * 1) ? 'yes' as const : 'no' as
const; // Ok, let b: "yes" | "no"
```

[37.4] Утверждение в сигнатуре (Signature Assertion)

Помимо функций, реализующих механизм *утверждения типа*, в *TypeScript* существует механизм *утверждения в сигнатуре*, позволяющий определять утверждающие функции, вызов которых, в случае невыполнения условия, приводит к выбрасыванию исключения. Для того, что бы объявить утверждающую функцию, в её сигнатуре (там где располагается возвращаемое значение) следует указать ключевое слово `asserts`, а затем параметр принимаемого на вход условия.

ts

```
function identifier(condition: any): asserts condition {
    if (!condition) {
        throw new Error('');
    }
}
```

Ключевой особенностью утверждения в сигнатуре является то, что в качестве аргумента утверждающая функция ожидает выражение, определяющие принадлежность к конкретному типу с помощью любого предназначенного для этого механизма (`typeof`, `instanceof` и даже с помощью механизма утверждения типов, реализуемого самим *TypeScript*).

Если принадлежность значения к указанному типу подтверждается, то далее по коду компилятор будет рассматривать его в роли этого типа. Иначе выбрасывается исключение.

ts

```ts
// утверждение в сигнатуре
function isStringAssert(condition: any): asserts condition {
    if (!condition) {
        throw new Error(``);
    }
}

// утверждение типа
function isString(value: any): value is string {
    return typeof value === 'string';
}

const testScope = (text: any) => {
    text.toUppercase(); // до утверждения расценивается как тип any..

    isStringAssert(text instanceof String); // выражение с оператором
instanceof
    isStringAssert(typeof text === 'string'); // выражение с оператором
typeof
    isStringAssert(isString(text)); // механизм "утверждения типа"

    text.toUppercase(); // ..после утверждения, как тип string
}
```

При использовании механизма *утверждения в сигнатуре* с механизмом *утверждения типа* , условие можно перенести из вызова утверждающей функции в её тело.

ts

```ts
function isStringAsserts(value: any): asserts value is string {
    if (typeof value !== "string") {
        throw new Error(``);
    }
}

const testScope = (text: any) => {
    text.toUppercase(); // не является ошибкой, потому, что тип — any

    isStringAsserts(text); // условие определено внутри утверждающей
функции

    text.toUppercase(); // теперь ошибка, потому, что тип утвержден как
string
}
```

Стоит обратить внимание на то, что механизм утверждения типа не будет работать в случае переноса условного выражения в тело утверждающей функции, сигнатура которой, лишена *утверждения типов* и содержит исключительно *утверждения в сигнатуре*.

ts

```ts
function isStringAsserts(value: any): asserts value /** is string */ {
    if (typeof value !== "string") {
        throw new Error(``);
    }
}

const testScope = (text: any) => {
    text.toUppercase(); // не является ошибкой, потому, что тип — any

    isStringAsserts(text); // условие определено в утверждающей функции

    text.toUppercase(); // нет ошибки, потому, что утверждение типов не
работает
}
```

Глава 38

Защитники типа

Понимание механизмов, рассматриваемых в этой главе, научит определять конструкции, которые часто применяются на практике и способны сделать код более понятным и выразительным.

[38.0] Защитники Типа - общее

Помимо того, что *TypeScript* имеет достаточно мощную систему выявления ошибок на этапе компиляции, разработчики языка, не останавливаясь на достигнутом, безостановочно работают над сведением их к нулю. Существенным шагом к достижению цели было добавление компилятору возможности активируемой при помощи флага `--strictNullChecks`, запрещающей неявные операции в которых участвует значение `null` и `undefined`. Простыми словами, компилятор научили во время анализа кода выявлять ошибки, способные возникнуть при выполнении операций, в которых фигурируют значения `null` или `undefined`.

Простейшим примером является операция получения элемента из dom-дерева при помощи метода `querySelector()`, который в обычном *нерекомендуемом* режиме (с неактивной опцией `--strictNullChecks`) возвращает значение, совместимое с типом `Element`.

ts

```
const stage: Element = document.querySelector('#stage');
```

Но в строгом *рекомендуемом* режиме (с активной опцией `--strictNullChecks`) метод `querySelector()` возвращает объединенный тип `Element | null`, поскольку искомое значение может попросту не существовать.

ts

```ts
const stage: Element | null = document.querySelector('#stage');
```

Не будет лишним напомнить, что на самом деле метод `querySelector` возвращает тип `Element | null` независимо от режима. Дело в том, что в обычном режиме тип `null` совместим с любыми типами. То есть, в случае отсутствия элемента в dom-дереве операция присваивания значения `null` переменной с типом `Element` не приведет к возникновению ошибки.

ts

```ts
// lib.es6.d.ts
interface NodeSelector {
    querySelector(selectors: string): Element | null;
}
```

Возвращаясь к примеру с получением элемента из dom-дерева стоит сказать, что в строке кода, в которой происходит подписка элемента на событие, на этапе компиляции все равно возникнет ошибка, даже в случае, если элемент существует. Дело в том, что компилятор *TypeScript* не позволит вызвать метод `addEventListener`, поскольку для него объект, на который ссылается переменная, принадлежит к типу `Element` ровно настолько же, насколько он принадлежит к типу `null`.

ts

```ts
const stage: Element | null = document.querySelector('#stage');
stage.addEventListener('click', stage_clickHandler); // тип переменной
stage Element или Null?

function stage_clickHandler(event: MouseEvent): void {}
```

Именно из-за этой особенности или другими словами, неоднозначности, которую вызывает тип `Union`, в *TypeScript*, появился механизм называемый *защитниками типа* (`Type Guards`).

Защитники типа — это правила, которые помогают выводу типов определить суженый диапазон типов для значения, принадлежащего к типу `Union`. Другими словами, разработчику предоставлен механизм, позволяющий с помощью выражений составить логические условия, проанализировав которые, вывод типов сможет сузить диапазон типов до указанного и выполнить над ним требуемые операции.

Понятно, что ничего не понятно. Поэтому, прежде чем продолжить разбирать определение по шагам, нужно рассмотреть простой пример, способный зафиксировать картинку в сознании.

Представим два класса, `Bird` и `Fish`, в обоих из которых реализован метод `voice`. Кроме этого, в классе `Bird` реализован метод `fly`, а в классе `Fish` — метод `swim`. Далее представим функцию с единственным параметром, принадлежащему к объединению типов `Bird` и `Fish`. В теле этой функции без труда получится выполнить операцию вызова метода `voice` у её параметра, так как этот метод объявлен и в типе `Bird`, и в типе `Fish`. Но при попытке вызвать метод `fly` или `swim` возникает ошибка, так как эти методы не являются общими для обоих типов. Компилятор попросту находится в подвешенном состоянии и не способен самостоятельно определится.

ts

```ts
class Bird {
    public fly(): void {}
    public voice(): void {}
}

class Fish {
    public swim(): void {}
    public voice(): void {}
}

function move(animal: Bird | Fish): void {
    animal.voice(); // Ok

    animal.fly(); // Error
    animal.swim(); // Error
}
```

Для того, что бы облегчить работу компилятору, *TypeScript* предлагает процесс сужения множества типов, составляющих тип `Union`, до заданного диапазона, а затем закрепляет его за конкретной областью видимости в коде. Но, прежде чем диапазон типов будет вычислен и ассоциирован с областью, разработчику необходимо составить условия, включающие в себя признаки, недвусмысленно указывающие на принадлежность к нужным типам.

Из-за того, что анализ происходит на основе логических выражений, область, за которой закрепляется суженый диапазон типов, ограничивается областью выполняемой при истинности условия.

Стоит заметить, что от признаков, участвующих в условии, зависит место, в котором может находится выражение, а от типов, составляющих множество типа `Union`, зависит способ составления логического условия.

[38.1] Сужение диапазона множества типов на основе типа данных

При необходимости составления условия, в основе которого лежат допустимые с точки зрения *JavaScript* типы, прибегают к помощи уже знакомых операторов как `typeof` и `instanceof`.

К помощи оператора `typeof` прибегают тогда, когда хотят установить принадлежность к типам `number`, `string`, `boolean`, `object`, `function`, `symbol` или `undefined`. Если значение принадлежит к производному от объекта типу, то установить его принадлежность к типу определяемого классом и находящегося в иерархии наследования, можно при помощи оператора `instanceof`.

Как уже было сказано, с помощью операторов `typeof` и `instanceof` составляется условие по которому компилятор может вычислить к какому конкретно типу или диапазону будет относиться значение в определяемой условием области.

ts

```ts
// Пример для оператора typeof


type ParamType = number | string | boolean | object | Function | symbol
| undefined;

function identifier(param: ParamType): void {
    param; // param: number | string | boolean | object | Function |
symbol | undefined

    if (typeof param === 'number') {
      param; // param: number
    } else if (typeof param === 'string') {
      param; // param: string
    } else if (typeof param === 'boolean') {
      param; // param: boolean
    } else if (typeof param === 'object') {
      param; // param: object
    } else if (typeof param === 'function') {
      param; // param: Function
    } else if (typeof param === 'symbol') {
      param; // param: symbol
    } else if (typeof param === 'undefined') {
      param; // param: undefined
    }
```

```
    param; // param: number | string | boolean | object | Function |
symbol | undefined
}
```

ts

```ts
// Пример для оператора instanceof


class Animal {
    constructor(public type: string) {}
}

class Bird extends Animal {}
class Fish extends Animal {}
class Insect extends Animal {}


function f(param: Animal | Bird | Fish | Insect): void {
    param; // param: Animal | Bird | Fish | Insect

    if (param instanceof Bird) {
        param; // param: Bird
    } else if (param instanceof Fish) {
        param; // param: Fish
    } else if (param instanceof Insect) {
        param; // param: Insect
    }

    param; // param: Animal | Bird | Fish | Insect
}
```

Если значение принадлежит к типу `Union`, а выражение состоит из двух операторов, `if` и `else`, значение находящиеся в операторе `else` будет принадлежать к диапазону типов не участвующих в условии `if`.

ts

```ts
// Пример для оператора typeof


function f0(param: number | string | boolean): void {
    param; // param: number | string | boolean

    if (typeof param === 'number' || typeof param === 'string') {
        param; // param: number | string
    } else {
        param; // param: boolean
    }

    param; // param: number | string | boolean
}
```

```
function f1(param: number | string | boolean): void {
    param; // param: number | string | boolean

    if (typeof param === 'number') {
        param; // param: number
    } else {
        param; // param: string | boolean
    }

    param; // param: number | string | boolean
}
```

ts

```
// Пример для оператора instanceof


class Animal {
    constructor(public type: string) {}
}

class Bird extends Animal {}
class Fish extends Animal {}
class Insect extends Animal {}
class Bug extends Insect {}

function f0(param: Bird | Fish | Insect): void {
    param; // param: Bird | Fish | Insect

    if (param instanceof Bird) {
        param; // param: Bird
    } else if (param instanceof Fish) {
        param; // param: Fish
    } else {
        param; // param: Insect
    }

    param; // param: Bird | Fish | Insect
}

function f1(param: Animal | Bird | Fish | Insect | Bug): void {
    param; // param: Animal | Bird | Fish | Insect | Bug
    if (param instanceof Bird) {
        param; // param: Bird
    } else if (param instanceof Fish) {
        param; // param: Fish
    } else {
        param; // param: Animal | Insect | Bug
    }

    param; // param: Animal | Bird | Fish | Insect | Bug
}
```

Кроме того, условия можно поместить в тернарный оператор. В этом случае область на которую распространяется сужение диапазона типов, ограничивается областью содержащей условное выражение.

Представьте функцию, которой в качестве единственного аргумента можно передать как значение, принадлежащее к типу `T`, так и функциональное выражение, возвращающее значение принадлежащие к типу `T`. Для того, что бы было проще работать со значением параметра, его нужно сохранить в локальную переменную, принадлежащую к типу `T`. Но прежде компилятору нужно помочь конкретизировать тип данных, к которому принадлежит значение.

Условие, как и раньше, можно было бы поместить в конструкцию `if` / `else`, но в таких случаях больше подходит тернарный условный оператор. Создав условие, в котором значение проверяется на принадлежность к типу, отличному от типа `T`, разработчик укажет компилятору, что при выполнении условия тип параметра будет ограничен типом `Function`, тем самым создав возможность вызвать параметр как функцию. Иначе значение, хранимое в параметре, принадлежит к типу `T`.

ts

```ts
// Пример для оператора typeof


function f(param: string | (() => string)): void {
  param; // param: string | (() => string)

  let value: string = typeof param !== 'string' ? param() : param;

  param; // param: string | (() => string)
}
```

ts

```ts
// Пример для оператора instanceof


class Animal {
    constructor(public type: string = 'type') {}
}

function identifier(param: Animal | (() => Animal)): void {
    param; // param: Animal | (() => Animal)

    let value: Animal = !(param instanceof Animal) ? param() : param;

    param; // param: Animal | (() => Animal)
}
```

Так как оператор `switch` логически похож на оператор `if` / `else`, то может показаться, что механизм, рассмотренный в этой главе, будет применим и к нему. Но это

не так. Вывод типов не умеет различать условия составленные при помощи операторов `typeof` и `instanceof` в конструкции `switch` .

[38.2] Сужение диапазона множества типов на основе признаков присущих типу Tagged Union

Помимо определения принадлежности к единичному типу, диапазон типов, составляющих тип `Union` , можно сузить по признакам, характерным для типа `Tagged Union` .

Условия, составленные на основе идентификаторов варианта, можно использовать во всех условных операторах включая `switch` .

ts

```ts
// Пример для оператора if/else


enum AnimalTypes {
    Animal = "animal",
    Bird = "bird",
    Fish = "fish"
}

class Animal {
    readonly type: AnimalTypes = AnimalTypes.Animal;
}

class Bird extends Animal {
    readonly type: AnimalTypes.Bird = AnimalTypes.Bird;

    public fly(): void {}
}

class Fish extends Animal {
    readonly type: AnimalTypes.Fish = AnimalTypes.Fish;

    public swim(): void {}
}

function move(param: Bird | Fish): void {
    param; // param: Bird | Fish
```

```
    if (param.type === AnimalTypes.Bird) {
        param.fly();
    } else {
        param.swim();
    }

    param; // param: Bird | Fish
}
```

ts

```
// Пример для тернарного оператора (?:)


function move(param: Bird | Fish): void {
    param; // param: Bird | Fish

    param.type === AnimalTypes.Bird ? param.fly() : param.swim();

    param; // param: Bird | Fish
}
```

ts

```
// Пример для оператора switch


enum AnimalTypes {
    Animal = "animal",
    Bird = "bird",
    Fish = "fish"
}

class Animal {
    readonly type: AnimalTypes = AnimalTypes.Animal;
}

class Bird extends Animal {
    readonly type: AnimalTypes.Bird = AnimalTypes.Bird;

    public fly(): void {}
}

class Fish extends Animal {
    readonly type: AnimalTypes.Fish = AnimalTypes.Fish;

    public swim(): void {}
}

function move(param: Bird | Fish): void {
    param; // param: Bird | Fish

    switch (param.type) {
```

```
        case AnimalTypes.Bird:
            param.fly(); // Ok
            break;

        case AnimalTypes.Fish:
            param.swim(); // Ok
            break;
    }

    param; // param: Bird | Fish
}
```

В случаях, когда множество типа `Union` составляют тип `null` и/или `undefined`, а также только один конкретный тип, выводу типов будет достаточно условия подтверждающего существование значения отличного от `null` и/или `undefined`. Это очень распространенный случай при активной опции `--strictNullChecks`. Условие, с помощью которого вывод типов сможет установить принадлежность значения к типам, отличными от `null` и/или `undefined`, может использоваться совместно с любыми условными операторами.

ts

```ts
// Пример с оператором if/else


function f(param: number | null | undefined): void {
    param; // param: number | null | undefined

    if (param !== null && param !== undefined) {
        param; // param: number
    }

    // or

    if (param) {
        param; // Param: number
    }

    param; // param: number | null | undefined
}
```

ts

```ts
// Пример с тернарным оператором (?:), оператором нулевого слияния (??,
nullish coalescing) и логическим "или" (||)


function f(param: number | null | undefined): void {
    param; // param: number | null | undefined

    var value: number = param ? param : 0;
    var value: number = param ?? 0;
```

```
    var value: number = param || 0;

    param; // param: number | null | undefined
}
```

ts

```
// Пример с оператором switch


function identifier(param: number | null | undefined): void {
    param; // param: number | null | undefined

    switch(param) {
        case null:
            param; // param: null
            break;

        case undefined:
            param; // param: undefined
            break;

        default: {
            param; // param: number
        }
    }

    param; // param: number | null | undefined
}
```

Кроме этого, при активной опции `--strictNullChecks`, в случаях со значением, принадлежащем к объектному типу, вывод типов может заменить оператор `Not-Null Not-Undefined`. Для этого нужно составить условие, содержащее проверку обращения к членам, в случае отсутствия которых может возникнуть ошибка.

ts

```
// Пример с Not-Null Not-Undefined (с учетом активной опции --
strictNullChecks)


class Ability {
    public fly(): void {}
}

class Bird {
    public ability: Ability | null = new Ability();
}

function move(animal: Bird | null | undefined): void {
    animal.ability // Error, Object is possibly 'null' or 'undefined'
    animal!.ability // Ok
```

```
    animal!.ability.fly(); // Error, Object is possibly 'null' or
'undefined'
    animal!.ability!.fly(); // Ok
}
```

ts

```ts
// Пример с защитником типа (с учетом активной опции --strictNullChecks)


class Ability {
    public fly(): void {}
}

class Bird {
    public ability: Ability | null = new Ability();
}

function move(animal: Bird | null | undefined): void {
    if (animal && animal.ability) {
        animal.ability.fly(); // Ok
    }

    // или с помощью оператора optional chaining
    if (animal?.ability) {
        animal.ability.fly(); // Ok
    }
}
```

[38.3] Сужение диапазона множества типов на основе доступных членов объекта

Сужение диапазона типов также возможно на основе доступных (`public`) членов, присущих типам, составляющим диапазон (`Union`). Сделать это можно с помощью оператора `in` .

ts

```ts
class A { public a: number = 10; }
class B { public b: string = 'text'; }
class C extends A {}


function f0(p: A | B) {
```

```
    if ('a' in p) {
        return p.a;  // p: A
    }

    return p.b;  // p: B
}

function f1(p: B | C) {
    if ('a' in p) {
        return p.a;  // p: C
    }

    return p.b;  // p: B
}
```

[38.4] Сужение диапазона множества типов на основе функции, определенной пользователем

Все перечисленные ранее способы работают только в том случае, если проверка происходит в месте отведенном под условие. Другими словами, с помощью перечисленных до этого момента способов, условие проверки нельзя вынести в отдельный блок кода (функцию). Это могло бы сильно ударить по семантической составляющей кода, а также нарушить принцип разработки программного обеспечения, который призван бороться с повторением кода (*Don't repeat yourself, DRY* (не повторяйся)). Но, к счастью для разработчиков, создатели *TypeScript* реализовали возможность определять пользовательские защитники типа.

В роли пользовательского защитника может выступать функция, функциональное выражение или метод, которые обязательно должны возвращать значения, принадлежащие к типу `boolean`. Для того, что бы вывод типов понял, что вызываемая функция не является обычной функцией, у функции вместо типа возвращаемого значения указывают предикат (предикат — это логическое выражение, значение которого может быть либо истинным `true`, либо ложным `false`).

Выражение предиката состоит из трех частей и имеет следующий вид `identifier is Type`.

Первым членом выражения является идентификатор, который обязан совпадать с идентификатором одного из параметров объявленных в сигнатуре функции. В случае, когда предикат указан методу экземпляра класса, в качестве идентификатора может быть указано ключевое слово `this`.

Стоит отдельно упомянуть, что ключевое слово `this` можно указать только в сигнатуре метода, определенного в классе или описанного в интерфейсе. При попытке указать ключевое слово `this` в предикате функционального выражения, не получится избежать ошибки, если это выражение определяется непосредственно в `prototype`, функции конструкторе, либо методе объекта, созданного с помощью литерала.

ts
```ts
// Пример с функцией конструктором


function Constructor() {}

Constructor.prototype.validator = function(): this is Object { // Error
    return true;
};
```
ts
```ts
// Пример с литералом объекта


interface IPredicat {
    validator(): this is Object; // Ok
}

var object: IPredicat = { // Ok
    validator(): this is Object { // Error
        return this;
    }
};

var object: {validator(): this is Object} = { // Error
    validator(): this is Object { // Error
        return this;
    }
};
```

Ко второму члену выражения относится ключевое слово `is`, которое служит в качестве утверждения. В качестве третьего члена выражения может выступать любой тип данных.

ts
```ts
// Пример предиката функции (function)


function isT1(p1: T1 | T2 | T3): p1 is T1 {
    return p1 instanceof T1;
}

function identifier(p1: T1 | T2 | T3): void {
```

```
    if (isT1(p1)) {
        p1; // p1: T1
    }
}
```

ts

```ts
// Пример предиката функционального выражения (functional expression)


const isT2 = (p1: T1 | T2 | T3): p1 is T2 => p1 instanceof T2;

function identifier(p1: T1 | T2 | T3): void {
    if (isT2(p1)) {
        p1; // p1: T2
    }
}
```

ts

```ts
// Пример предиката метода класса (static method)


class Validator {
    public static isT3(p1: T1 | T2 | T3): p1 is T3 {
        return p1 instanceof T3;
    }
}

function identifier(p1: T1 | T2 | T3): void {
    if (Validator.isT3(p1)) {
        p1; // p1: T3
    }
}
```

Условие, на основании которого разработчик определяет принадлежность одного из параметров к конкретному типу данных, не ограничено никакими конкретными правилами. Исходя из результата выполнения условия `true` или `false`, вывод типов сможет установить принадлежность указанного параметра к указанному типу данных.

ts

```ts
class Animal {}
class Bird extends Animal {
    public fly(): void {}
}

class Fish extends Animal {
    public swim(): void {}
}

class Insect extends Animal {
    public crawl(): void {}
```

```
}

class AnimalValidator {
    public static isBird(animal: Animal): animal is Bird {
        return animal instanceof Bird;
    }

    public static isFish(animal: Animal): animal is Fish {
        return (animal as Fish).swim !== undefined;
    }

    public static isInsect(animal: Animal): animal is Insect {
        let isAnimalIsNotUndefinedValid: boolean = animal !== undefined;
        let isInsectValid: boolean = animal instanceof Insect;

        return isAnimalIsNotUndefinedValid && isInsectValid;
    }
}

function move(animal: Animal): void {
    if (AnimalValidator.isBird(animal)) {
        animal.fly();
    } else if (AnimalValidator.isFish(animal)) {
        animal.swim();
    } else if (AnimalValidator.isInsect(animal)) {
        animal.crawl();
    }
}
```

Последнее, о чем осталось упомянуть, что в случае, когда по условию значение не подходит ни по одному из признаков, вывод типов установит его принадлежность к типу `never`.

ts

```
class Animal {
    constructor(public type: string) {}
}

class Bird extends Animal {}
class Fish extends Animal {}

function move(animal: Bird | Fish): void {
    if (animal instanceof Bird) {
        animal; // animal: Bird
    } else if (animal instanceof Fish) {
        animal; // animal: Fish
    } else {
        animal; // animal: never
    }
}
```

Глава 39

Вывод типов

Понимание темы, относящейся к такому фундаментальному механизму как *вывод типов*, поможет разработчику подчинить компилятор *tsc*, а не наоборот. Невозможно писать программы на языке *TypeScript*, получая от процесса удовольствие, если нет однозначного ответа на вопрос "указывать типы явно или нет". Ответы на этот и другие сопряженные вопросы, как раз и содержит данная глава, посвященная подробному рассмотрению каждого момента.

[39.0] Вывод типов - общие сведения

Чтобы не повторять определения, которые были даны в главе “Экскурс в типизацию - Связывание, типизация, вывод типов”, эту главу стоит начать с неформально определения.

Вывод типов — это механизм, позволяющий сделать процесс разработки на статически типизированном *TypeScript* более простой, за счет перекладывания на него рутинной работы по явной аннотации типов. Может показаться, что вывод типов берется за дело только тогда, когда при разборе кода попадается отсутствующая аннотация типа. Но это не так. Компилятор не доверяет разработчику и весь код, в штатном режиме, проходит через вывод типов. Не важно, в полной мере разработчик указывает типы определяемым им конструкциям или нет, что бы их проверить на адекватность и совместимость, вывод типов обязан создать для них собственное описание (понимать как объявление типа).

В этом механизме нет ничего сложного, но, несмотря на это, у начинающих разработчиков *TypeScript*, некоторые неочевидные особенности могут вызвать вопросы, главным из которых является уже упомянутый в самом начале - "указывать явно типы или нет".

Ответ очевиден и прост — во всех случаях, допускающих отсутствие явной аннотации типа, эту работу стоит поручать выводу типов. Другими словами, не нужно указывать типы явно, если это за вас сможет сделать вывод типов. Единственное исключение может возникнуть при необходимости повышения семантики кода, что относительно аннотаций типа бывает сравнительно редко.

[39.1] Вывод примитивных типов

Вывод типов, для значений принадлежащих к так называемым примитивным типам, не таит в себе ничего необычного. Кроме того, будь это переменные, поля, параметры, возвращаемые из функций и методов, или значения — результат во всех случаях будет идентичным.

ts

```ts
enum Enums {
    Value
};

let v0 = 0; // let v0: number
let v1 = 'text'; // let v1: string
let v2 = true; // let v2: boolean
let v3 = Symbol(); // let v3: symbol
let v4 = Enums.Value; // let v4: Enums
```

[39.2] Вывод примитивных типов для констант (const) и полей только для чтения (readonly)

Когда дело доходит до присваивания значений, принадлежащих к примитивным типам, таким конструкциям, как константы (`const`) и неизменяемые поля (модификатор `readonly`), поведение вывода типов изменяется.

В случае, когда значение принадлежит к примитивным типам `number` , `string` или `boolean` , вывод типов указывает принадлежность к литеральным примитивным типам, определяемым самим значением.

ts

```ts
const v0 = 0; // let v0: 0
const v1 = 'text'; // let v1: 'text'
const v2 = true; // let v2: true

class Identifier {
    readonly f0 = 0; // f0: 0
    readonly f1 = 'text'; // f1: 'text'
    readonly f2 = true;// f2: true
}
```

Если значение принадлежит к типу `enum` , то вывод типов установит принадлежность к указанному значению.

ts

```ts
enum Enums {
    Value
}

const v = Enums.Value; // const v: Enums.Value

class Identifier {
    readonly f = Enums.Value; // f: Enums.Value
}
```

Когда вывод типов встречает значение, принадлежащее к типу `symbol` , его поведение зависит от конструкции, которой присваивается значение. Так, если вывод типов работает с константой, то тип определяется как запрос типа (глава “Type Queries (запросы типа), Alias (псевдонимы типа)”) самой константы. Если вывод типов устанавливает

принадлежность к типу неизменяемого поля, то тип будет определен как `symbol`. Происходит так потому, что вместе с созданием нового экземпляра в системе будет определяться и новый символ, что противоречит правилам установленным для `Unique Symbol` (глава “Типы - Примитивные литеральные типы Number, String, Boolean, Unique Symbol, Enum”).

ts

```ts
const v = Symbol(); // const v: typeof v

class Identifier {
    readonly f = Symbol(); // f: symbol
}
```

[39.3] Вывод объектных типов

С выводом объектных типов не связанно ничего необычного. Кроме того, поведение вывода типов одинаково для всех конструкций.

ts

```ts
class ClassType {}
interface InterfaceType {}

type TypeAlias = number;

let typeInterface: InterfaceType;
let typeTypeAlias: TypeAlias;

let v0 = { a: 5, b: 'text', c: true }; // let v0: {a: number, b:string,
c: boolean}
const v1 = { a: 5, b: 'text', c: true }; // const v1: {a: number, b:
string, c: boolean}

let v3 = new ClassType(); // let v3: ClassType
let v4 = typeInterface; // let v4: InterfaceType
let v5 = typeTypeAlias; // let v5: number
```

[39.4] Вывод типа для полей класса на основе инициализации их в конструкторе

Если прочитать главу, посвященную конфигурации компилятора, станет известно, что при активном флаге `--noImplicitAny`, возникает ошибка, если тело класса включает поля без аннотации типа. Дело в том, что вывод типов расценивает поля без явной аннотации типа как принадлежащие к `any`, который как раз и не допускает активированный флаг `--noImplicitAny`.

ts

```
class Square {
    /**
     * Поля без явной аннотации типа.
     * Вывод типов определяет их
     * принадлежность к типу any.
     *
     * (property) Square.area: any
     * (property) Square.sideLength: any
     *
     * От этого возникает ошибка ->
     * TS7008: Member 'area' implicitly has an 'any' type.
     * TS7008: Member 'sideLength' implicitly has an 'any' type.
     */
    area;
    sideLength;

    constructor() {
    }
}
```

Но, к счастью, тип полей без явной аннотации может быть автоматически выведен, если инициализация таких полей происходит в конструкторе.

ts

```
class Square {
    /**
     * Поля без явной аннотации типа,
     * но ошибки не возникает, поскольку
     * вывод типов определяет их принадлежность
     * к типу number, так как поле sideLength
     * инициализируется в конструкторе его параметром,
```

```
     * принадлежащим к типу number, а поле area инициализируется
     * там же с помощью выражения, результат которого
     * также принадлежит к типу number.
     *
     * (property) Square.area: number
     * (property) Square.sideLength: number
     */
    area;
    sideLength;

    constructor(sideLength: number) {
        this.sideLength = sideLength;
        this.area = this.sideLength ** 2;
    }
}
```

Не будет лишним сделать акцент на словах об инициализации в конструкторе, поскольку это условие является обязательным. При попытке инициализации полей вне тела конструктора будет вызвана ошибка, даже если инициализация производится в методе, вызываемом из конструктора.

ts

```
class Square {
    /**
     * Error ->
     * TS7008: Member 'area' implicitly has an 'any' type.
     */
    area;
    sideLength;

    constructor(sideLength: number) {
        this.sideLength = sideLength;
        this.init();
    }

    init(){
        this.area = this.sideLength ** 2;
    }
}
```

Если инициализация полей класса без аннотации по каким-то причинам может не состоятся, то тип будет выведен как объединение, включающее так же и тип `undefined` .

ts

```
class Square {
    /**
     * [1] ...вывод типов определяет принадлежность
     * поля sideLength как ->
```

```
     *
     * (property) Square.sideLength: number | undefined
     */
    sideLength;

    constructor(sideLength: number) {
        /**
         * [0] Поскольку инициализация зависит от
         * условия выражения, которое выполнится
         * только во время выполнения программы...
         */
        if (Math.random()) {
            this.sideLength = sideLength;
        }
    }

    get area() {
        /**
         * [2] Тем не менее, возникает ошибка,
         * поскольку операция возведения в степень
         * производится над значением, которое может
         * быть undefined
         *
         * Error ->
         * Object is possibly 'undefined'.
         */
        return this.sideLength ** 2;
    }
}
```

[39.5] Вывод объединенных (Union) типов

С выводом типов объединения (глава “Типы - Union, Intersection”) связаны как очевидные, так и нет, случаи.

К очевидным случаям можно отнести массив, состоящий из разных примитивных типов. В этом случае будет выведен очевидный тип *объединение*, которое определяется типами присутствующих в массиве примитивов.

ts

```
let v = [0, 'text', true]; // let v: (string | number | boolean)[]
```

В случае получения любого элемента массива, вывод типов также установит принадлежность к объединенному типу.

ts

```ts
let v = [0, 'text', true]; // let v: (string | number | boolean)[]

let item = v[0]; // let item: string | number | boolean
```

Неочевидные особенности лучше всего начать с примера, в котором вывод типа определяет принадлежность значения к массиву, состоящему из обычных объектных типов.

ts

```ts
let v = [
    { a: 5, b: 'text' },
    { a: 6, b: 'text' }
]; // let v: { a: number, b: string }[]
```

В примере, вывод типов выводит ожидаемый и предсказуемый результат для массива объектов, чьи типы полностью идентичны. Идентичны они по той причине, что вывод типов установит тип `{a: number, b: string}` для всех элементов массива.

Но стоит изменить условие, допустим, убрать объявление одного поля и картина кардинально изменится. Вместо массива обычных объектов, тип будет выведен, как массив объединенного типа.

ts

```ts
let v = [
    { a: 5, b: 'text' },
    { a: 6 },
    { a: 7, b: true }
]; // let v: ({ a: number, b: string } | { a: number, b?: undefined } |
{ a: number, b: boolean })[]
```

Как видно из примера выше, вывод типов приводит все объектные типы, составляющие тип объединение, к единому виду. Он добавляет к типам несуществующие в них, но существующие в других объектных типах, поля, декларируя их как необязательные (глава “Операторы - Optional, Not-Null, Not-Undefined, Definite Assignment Assertion”). Сделано это для возможности конкретизировать тип любого элемента массива. Простыми словами, что бы не получить ошибку во время выполнения, любой элемент массива должен иметь общие для всех элементов признаки. Но так как в реальности, в объектах, некоторые члены вовсе могут отсутствовать, вывод типов, что бы повысить типобезопасность, декларирует их как необязательные. Таким образом он предупреждает разработчика о возможности возникновения ситуации, при которой эти члены будут иметь значение `undefined`, что и демонстрируется в примере ниже.

ts

```ts
let v = [
    { a: 5, b: 'text' },
    { a: 6 },
    { a: 7, b: true }
]; // let v: ({ a: number, b: string } | { a: number, b?: undefined } |
{ a: number, b: boolean })[]

let a = v[0].a; // let a: number
let b = v[0].b; // let b: string | boolean | undefined
```

Если в качестве значений элементов массива выступают экземпляры классов, не связанных отношением наследования, то они и будут определять тип объединение.

ts

```ts
class A {
    public a: number = 0;
}

class B {
    public a: string = '';
    public b: number = 5;
}

let v = [
    new A(),
    new B()
]; // let v: (A | B)[]
```

В случае, если элементы массива являются экземплярами классов, связанных отношением наследования (*номинативная типизация* “Экскурс в типизацию - Совместимость типов на основе вида типизации”), то выводимый тип будет ограничен самым базовым типом.

ts

```ts
class A {}
class B extends A { f0 = 0; }
class C extends A { f1 = ''; }
class D extends A { f2 = true; }
class E extends D { f3 = {}; }

let v3 = [new A(), new B(), new C(), new D(), new E()];  // let v3: A[]
let v4 = [new B(), new C(), new D(), new E()];  // let v4: (B | C | D)[]
```

Те же самые правила применяются при выводе типа значения, возвращаемого тернарным оператором.

ts

```ts
class A {}
class B extends A { f0 = 0; }
class C extends A { f1 = ''; }
class D extends A { f2 = true; }
class E extends D { f3 = {}; }

let v0 = false ? new A() : new B();  // let v0: A
let v1 = false ? new B() : new C();  // let v1: B | C
let v2 = false ? new C() : new D();  // let v2: C | D
```

Так как выражение, расположенное в блоке тернарного оператора, вычисляется на этапе выполнения программы, вывод типов не может знать результата его вычисления на этапе компиляции. Поэтому, что бы не нарушить типобезопасность, он вынужден указывать объединенный тип, определяемый всеми блоками выражения.

[39.6] Вывод пересечения (Intersection) с дискриминантными полями

Если при определении типа пересечения, определяющее его множество включает больше одного типа, определяющего одноименные дискриминантные поля, принадлежащие к разным типам, то такое пересечение определяется как тип `never`. Данная тема подробно бала рассмотрены в главе "Типы - Discriminated Union"

ts

```ts
type A = {
    type: "a"; // дискриминантное поле

    a: number;
};
type B = {
    type: "b"; // дискриминантное поле

    b: number;
};
type C = {
    c: number;
};
type D = {
    d: number;
};
```

```ts
/**
 * Как видно, типы A и B
 * определяют одноименное
 * дискриминантное поле type,
 * принадлежащее к разным типам "a" и "b",
 * поэтому тип T будет определен
 * как тип never.
 *
 * type T = never
 */

type T = A & B & C & D;
```

Но стоит обратить внимание, что речь идет только об одноименных полях принадлежащих к разным типам. То есть, если множество, определяющее пересечение, включает несколько типов с одноименными дискриминантными полями, принадлежащих к одному типу, то такое множество будет определено ожидаемым образом.

ts

```ts
type A = {
    type: "a"; // дискриминантное поле с типом a

    a: number;
};
type B = {
    type: "a"; // дискриминантное поле с типом b

    b: number;
};
type C = {
    c: number;
};
type D = {
    d: number;
};

/**
 * Как видно, типы A и B
 * по-прежнему определяют одноименные
 * дискриминантные поля type, но на этот
 * раз они принадлежат к одному типу "a",
 * поэтому тип T будет определен ожидаемым
 * образом.
 *
 *
 *
 * type T = A & B & C & D
 */
```

```ts
type T = A & B & C & D;
```

[39.7] Вывод типов кортеж (Tuple)

Начать стоит с напоминания, что значение длины кортежа, содержащего элементы, помеченные как необязательные, принадлежит к типу объединению (`Union`), который составляют литеральные числовые типы.

ts

```ts
function f(...rest: [number, string?, boolean?]): [number, string?,
boolean?] {
    return rest;
}

let l = f(5).length; // let l: 1 | 2 | 3
```

Кроме того, остаточные параметры (`...rest`), аннотированные с помощью параметра типа, рассматриваются и представляются выводом типа как принадлежащие к типу-кортежу.

ts

```ts
function f<T extends any[]>(...rest: T): T {
    return rest;
}

// рассматриваются

f(5); // function f<[number]>(rest_0: number): [number]
f(5, ''); // function f<[number, string]>(rest_0: number, rest_1:
string): [number, string]
f(5, '', true); // function f<[number, string, boolean]>(rest_0:
number, rest_1: string, rest_2: boolean): [number, string, boolean]

// представляются

let v0 = f(5); // let v0: [number]
let v1 = f(5, ''); // let v1: [number, string]
let v2 = f(5, '', true); // let v2: [number, string, boolean]
```

Если функция, определяющая остаточные параметры, принадлежащие к параметру типа, будет вызвана с аргументами, включающих массив, указанный при помощи расширяющего синтаксиса (*spread syntax*), то тип для него будет выведен в виде остаточного типа `...rest` .

ts

```ts
function tuple<T extends any[]>(...args: T): T {
    return args;
}

let numberAll: number[] = [0, 1, 2];
let v0 = tuple(5, '', true); // let v0: [number, string, boolean]
let v1 = tuple(5, ...numberAll); // let v1: [number, ...number[]]
let v2 = tuple(5, ...numberAll, ''); // let v2: [number, ...number[],
string]
```

Глава 40

Совместимость объектных типов (Compatible Object Types)

На практике очень много недопониманий связано с темой совместимости объектных типов, постижение которой возможно лишь путем последовательного рассмотрения каждого отдельного случая в разных ситуациях. Именно этому и будет посвящена текущая глава, которая также уделит немало внимания другим сопутствующим нюансам.

[40.0] Важно

Пришло время более подробно разобраться как компилятор определяет совместимость объектных типов. Как всегда, вначале, стоит напомнить, что в текущей главе, будет использоваться шаблон (`: Target = Source`), о котором более подробно шла речь в самом начале.

Но, прежде чем начать погружение в тему *совместимости типов* (*compatible types*), будет не лишним заметить, что подобный термин не определен спецификацией *TypeScript*. Тем не менее, в *TypeScript* описано два типа совместимости. Помимо привычной *совместимости подтипов* (*assignment subtype*), также существует *совместимость при присваивании* (*assignment compatibility*). Они отличаются только тем, что правила совместимости при присваивании расширяют правила совместимости подтипов. Сделано это по нескольким причинам.

Прежде всего поведение типа `any` не укладывается в рамки, определяемые стандартными правилами. Нестандартное поведение заключается в том, что помимо

совместимости всех типов на основе обычных правил совместимости с типом `any`, сам тип `any` также совместим со всеми не являясь их подтипом.

ts

```ts
class Animal {
    public name: string;
}

class Bird extends Animal {
    public fly(): void {}
}

let animal: Animal = new Bird(); // Ok
let bird: Bird = new Animal(); // Ошибка, присваивание подтипа

let any: any = 0; // Ok
let number: number = any; // Ok -> any совместим с number
```

Кроме того, поведением двухсторонней совместимости наделен и числовой `enum`.

ts

```ts
enum NumberEnum {
    A = 1
}

let v1: number = NumberEnum.A;
let v2: NumberEnum.A = 0;
```

[40.1] Совместимость объектных типов в TypeScript

Начать тему о совместимости объектных типов стоит с повторения определения структурной типизации, которая лежит в основе *TypeScript*. Итак, *структурная типизация* — это механизм сопоставления двух типов на основе их признаков. Под признаками понимаются идентификаторы типа и типы, которые с ними связаны (ассоциированы).

Простыми словами, два типа будут считаться совместимыми не потому, что они связаны иерархическим деревом (наследование), а по тому, что в типе `S` (`: Target = Source`) присутствуют все идентификаторы, присутствующие в типе `T`. При этом, типы, с которыми они ассоциированы, должны быть совместимы.

ts

```ts
class Bird {
    public name: string;
}

class Fish {
    public name: string;
}

let bird: Bird;
let fish: Fish;

let v1: Bird = fish; // Ok
let v2: Fish = bird; // Ok
```

В случаях, когда один тип, помимо всех признаков второго типа, также имеет любые другие, то он будет совместим со вторым типом, но не наоборот. Для обратной совместимости потребуется операция явного преобразования (приведения) типов.

ts

```ts
class Bird {
    public name: string;
    public age: number;
}

class Fish {
    public name: string;
}

var bird: Bird = new Fish(); // Error
var bird: Bird = new Fish() as Bird; // Ok
let fish: Fish = new Bird(); // Ok
```

Кроме того, два типа, совместимые по признакам идентификаторов, будут совместимы только в том случае, если типы, ассоциированные с идентификаторами, также совместимы.

ts

```ts
class Bird {
    public name: string;
    public age: number;
}

class Fish {
    public name: string;
}

class BirdProvider {
```

```
    public data: Bird;
}

class FishProvider {
    data: Fish;
}

let birdProvider: BirdProvider = new FishProvider(); // Error
let birdProvider: BirdProvider = new FishProvider() as FishProvider; //
Error
let fishProvider: FishProvider = new BirdProvider(); // Ok
```

Стоит заметить, что методы, объявленные в объектном типе, сравниваются не по правилам совместимости объектных типов данных. Про правила проверки функциональных типов речь пойдет немного позднее (глава “Типизация - Совместимость функций”). Поэтому комментарии к коду будут опущены.

ts

```
class Bird {
    public voice(repeat: number): void {}
}

class Fish {
    public voice(repeat: number, volume: number): void {}
}

let v1: Bird;
let v2: Fish;

let v3: Bird = v2; // Error
let v4: Fish = v1; // Ok
```

По этой же причине без подробного рассмотрения останется и следующий пример, в котором происходит проверка типов, содержащих перегруженные методы, поскольку их совместимость идентична совместимости функциональных типов, рассматриваемых в следующей главе. Сейчас стоит только упомянуть, что в случаях, когда функция перегружена, проверка на совместимость происходит для каждой из сигнатур. Если существует несколько вариантов перегруженных сигнатур, с которыми может быть совместим тип источник, то выбрана будет та, что объявлена раньше.

ts

```
class Bird {
    public voice(repeat: number, volume: number): void;
    public voice(repeat: number): void {}
}

class Fish {
    public voice(repeat: number, volume: number): void {}
}
```

```
let v1: Bird
let v2: Fish

let v3: Bird = v2; // Ok
let v4: Fish = v1; // Ok
```

Типы, которые различаются только *необязательными членами*, также считаются совместимыми.

ts

```ts
class Bird {
    public name: string;
    public age?: number;

    public fly?(): void {}
}

class Fish {
    public name: string;
    public arial?: string;

    public swim?(): void {}
}

let bird: Bird;
let fish: Fish;

// class Bird {name: string} === class Fish {name: string}

let v1: Bird = fish; // Ok
let v2: Fish = bird; // Ok
```

Дело в том, что необязательные параметры в объектных типах не берутся в расчет при проверке на совместимость. Однако это правило действует только в одну сторону. Тип содержащий обязательный член, несовместим с типом, у которого идентичный член является необязательным. Такое поведение логично, ведь в случае, когда необязательный член будет отсутствовать, тип, содержащий его, не будет удовлетворять условиям, заданным типом с обязательным членом.

ts

```ts
class Bird {
    public name: string;
    public age?: number;
}

class Fish {
    public name: string;
    public age:  number;
```

```
}

let bird: Bird;
let fish: Fish;
/**
* Bird -> name -> поиск в Fish -> член найден -> Fish['name'] -> Ok
* Bird -> age -> age опциональный член -> пропуск
*/
let v1: Bird = fish; // Ok
/**
* Fish -> name -> поиск в Bird -> член найден -> Bird['name'] -> Ok
* Fish -> age -> поиск в Bird -> член найден -> Bird['age'] не является
опциональным -> Ошибка
*/
let v2: Fish = bird; // Error
let v3: Fish = bird as Fish; // Ok
```

Существует еще одна неочевидность, связанная с необязательными членами. Если в целевом типе все члены объявлены как необязательные, он будет совместим с любым типом, который частично описывает его, при этом тип-источник может описывать любые другие члены. Помимо этого он будет совместим с типом, у которого описание отсутствует вовсе. Но он не будет совместим с типом, у которого описаны только отсутствующие в целевом типе члены. Такое поведение в *TypeScript* называется *Weak Type Detection* (*обнаружение слабого типа*). Типы, описание которых состоит только из необязательных членов, считаются слабыми типами.

ts

```
class IAnimal {
    name?: string;
    age?: number;
}

class Animal {}
class Bird { name: string; }
class Fish { age: number; }
class Insect { name: string; isAlive: boolean; }
class Reptile { age: number; isAlive: boolean; }
class Worm { isAlive: boolean; }

let animal: Animal;
let bird: Bird;
let fish: Fish;
let insect: Insect;
let reptile: Reptile;
let worm: Worm;

let v1: IAnimal = animal; // Ok
let v2: IAnimal = bird; // Ok
let v3: IAnimal = fish; // Ok
let v4: IAnimal = insect; // Ok
```

```
let v5: IAnimal = reptile; // Ok
let v6: IAnimal = worm; // Error
```

Обобщенные типы, закрытые частично или полностью, участвуют в проверке на совместимость по характерным для *TypeScript* правилам.

ts

```
class Bird<T> {
    public name: T;
}

class Fish<T, S> {
    public name: T;
    public age: S;
}

let v1: Bird<string>;
let v2: Bird<number>;

let v3: Bird<string> = v2; // Error
let v4: Bird<number> = v1; // Error

let v5: Bird<string>;
let v6: Fish<string, number>;

let v7: Bird<string> = v6; // Ok
let v8: Fish<string, number> = v5; // Error
```

В случаях, когда на совместимость проверяются типы, содержащие обобщенные методы, то их сравнение ничем не отличается от сравнения типов, содержащих необобщенные методы.

ts

```
class Bird {
    public voice<T>(repeat: T): void {}
}

class Fish {
    public voice<T, S>(repeat: T, volume: S): void {}
}

let v1: Bird
let v2: Fish

let v3: Bird = v2; // Error
let v4: Fish = v1; // Ok
```

На фоне структурной типизации самое неоднозначное поведение возникает, когда описание типов полностью идентично, за исключением их модификаторов доступа. Если в типе описан хоть один член с отличным от `public` модификатором доступа, он не будет совместим ни с одним схожим типом, независимо от того, какие модификаторы доступа применены к его описанию.

ts

```ts
class Bird {
    private name: string;
}

class Fish {
    private name: string;
}

class Insect {
    protected name: string;
}

class Reptile {
    public name: string;
}

let v1: Bird;
let v2: Fish;
let v3: Insect;
let v4: Reptile;

let v5: Bird = v2; // Error
let v6: Fish = v1; // Error
let v7: Insect = v1; // Error
let v8: Reptile = v1; // Error
```

К счастью, разногласия, возникающие в структурной типизации при совместимости типов, представляемых классами, к членам которых применены модификаторы доступа, отличные от `public`, не распространяются на номинативную типизацию (глава “Экскурс в типизацию - Совместимость типов на основе вида типизации”). Номинативная типизация может указывать на принадлежность к типу через иерархию наследования. Простыми словами, потомки будут совместимы с предками, у которых члены объявлены с помощью модификаторов доступа, отличных от `public`.

ts

```ts
class Bird {
    protected name: string;
}

class Owl extends Bird {
    protected name: string;
}
```

```
let bird: Bird;
let owl: Owl;

let v1: Bird = owl; // Ok
let v2: Owl = bird; // Error
let v3: Owl = bird as Owl; // Ok
```

В типах, определяемых классами, при проверке на совместимость не учитываются конструкторы и статические члены (члены класса).

ts

```
class Bird {
    public static readonly DEFAULT_NAME: string = 'bird';

    constructor(name: string) {}
}

class Fish {
    public static toStringDecor(target: string): string {
        return `[object ${target}]`;
    }

    constructor(age: number) {}
}

let v1: Bird
let v2: Fish

let v3: Bird = v2; // Ok
let v4: Fish = v1; // Ok
```

Когда в качестве присваиваемого типа выступает экземпляр класса, то для того, что бы он считался совместим с типом, указанным в аннотации, в нем как минимум должны присутствовать все признаки этого типа. Также он может обладать дополнительными признаками, которые отсутствуют в типе указанном в аннотации.

ts

```
class Bird {
    public name: string;
}

class Fish {
    public name: string;
    public age: number;
}

class Insect {}

let equal: Bird = new Bird();
```

```
let more: Fish = new Fish();
let less: Insect = new Insect();

interface IAnimal {
    name: string;
}

let v1: IAnimal = new Bird(); // Ok -> одинаковые поля
let v2: IAnimal = new Fish(); // Ok -> в Fish полей больше
let v3: IAnimal = new Insect(); // Ошибка -> обязательные поля
отсутствуют
let v4: IAnimal = equal; // Ok -> одинаковые поля
let v5: IAnimal = more; // Ok -> в Fish полей больше
let v6: IAnimal = less; // Ошибка -> обязательные поля отсутствуют

function f1(p1: IAnimal): void {}

f1(new Bird()); // Ok -> одинаковые поля
f1(new Fish()); // Ok -> в Fish полей больше
f1(new Insect()); // обязательные поля отсутствуют

f1(equal); // Ok -> одинаковые поля
f1(more); // Ok -> в Fish полей больше
f1(less); // обязательные поля отсутствуют
```

Однако, когда в качестве значения выступает объектный тип, созданный с помощью объектного литерала, поведение в некоторых случаях отличается от поведения присвоения экземпляров класса. В тех случаях, в которых объект объявляется непосредственно в операции присвоения, он будет совместим с типом, указанным в аннотации только, если он полностью ему соответствует. Другими словами, создаваемый с помощью литерала объект не должен содержать ни меньше ни больше членов, чем описано в типе указанном в аннотации (данное поведение можно изменить с помощью опции компилятора `--suppressExcessPropertyErrors` , глава “Опции компилятора”).

ts

```
interface IAnimal {
    name: string;
}

function f1(p1: IAnimal): void {}

let equal = { name: '' };
let more = { name: '', age: 0 };
let less = {};

var v1: IAnimal = { name: '' }; // Ok -> одинаковые поля
let v2: IAnimal = { name: '', age: 0 }; // Ошибка-> полей больше
let v3: IAnimal = {}; // Ошибка -> полей меньше

let v4: IAnimal = equal; // Ok -> одинаковые поля
```

```
let v5: IAnimal = more; // Ok -> полей больше
let v6: IAnimal = less; // Ошибка -> полей меньше

f1({ name: '' }); // Ok -> одинаковые поля
f1({ name: '', age: 0 }); // Ошибка -> больше полей
f1({}); // Ошибка -> полей меньше

f1(equal); // Ok -> одинаковые поля
f1(more); // Ok -> полей больше
f1(less); // Ошибка -> полей меньше
```

Остается только добавить, что выбор в сторону структурной типизации был сделан по причине того, что подобное поведение очень схоже с поведением самого *JavaScript*, который реализует *утиную типизацию*. Можно представить удивление *Java* или *C#* разработчиков, которые впервые увидят структурную типизацию на примере *TypeScript*. Сложно представить выражение лица заядлых теоретиков, когда они увидят, что сущность птицы совместима с сущностью рыбы. Но не стоит угнетать ситуацию, выдумывая нереальные примеры, которые из-за структурной типизации приведут к немыслимым последствиям, поскольку вероятность того, что хотя бы один из них найдет олицетворение в реальных проектах настолько мала, что не стоит сил затраченных на их выдумывание.

Глава 41

Совместимость функциональных типов (Compatible Function Types)

После объектных типов, мир совместимости функциональных типов может показаться перевернутым с ног на голову. Он обладает множеством нюансов, каждый из которых будет детально рассмотрен в текущей главе.

[41.0] Важно

Важной частью работы с функциями является понимание совместимости функциональных типов. Поверхностное понимание механизма совместимости функциональных типов может сложить ошибочное чувство их постижения, поскольку то, что на первый взгляд может казаться очевидным, не всегда может являться таковым. Для того, что бы понять замысел создателей *TypeScript*, нужно детально разобрать каждый момент. Но прежде стоит уточнить одну деталь. В примерах, которые будут обсуждаться в главе, посвященной типизации функциональных типов, будет использоваться уточняющий шаблон `: Target = Source`. Кроме того, объектные типы, указанные в сигнатуре функции, ведут себя так же, как было описано в главе, посвященной совместимости объектных типов.

[41.1] Совместимость параметров

Первое, на, что стоит обратить внимание, это параметры функции. На параметры функции приходится наибольшее количество неоднозначностей, связанных с совместимостью.

Начать стоит с того, что две сигнатуры считаются совместимыми, если они имеют равное количество параметров с совместимыми типами данных.

ts

```ts
type T1 = (p1: number, p2: string) => void;

let v1: T1 = (p3: number, p4: string) => {}; // Ok -> разные
идентификаторы
let v2: T1 = (p1: number, p2: boolean) => {}; // Error
```

При этом стоит заметить, что идентификаторы параметров не участвуют в проверке на совместимость.

ts

```ts
type T1 = (...rest: number[]) => void;

let v1: T1 = (...numbers: number[]) => {}; // Ok -> разные
идентификаторы
```

Кроме того, параметры, помеченные как необязательные, учитываются только тогда, когда они участвуют в проверке на совместимость по признакам количества параметров.

ts

```ts
type T1 = (p1: number, p2?: string) => void;

let v1: T1 = (p1: number) => {}; // Ok
let v2: T1 = (p1: number, p2: string) => {}; // Ok или Error с
включенным флагом --strictNullChecks
let v3: T1 = (p1: number, p2: boolean) => {}; // Error
let v4: T1 = (p1: number, p2?: boolean) => {}; // Error
```

Функция, имеющая определение остаточных параметров, будет совместима в обе стороны с любой функцией, так как остаточные параметры расцениваются способными принадлежать к любому типу и чье количество находится в диапазоне от нуля до бесконечности.

ts

```ts
type T1 = (...rest: any[]) => void;
type T2 = (p0: number, p1: string) => void;

let v0: T1 = (...rest) => {};
let v1: T2 = (p0, p1) => {};

let v2: T1 = v1; // Ok
let v3: T2 = v0; // Ok
```

В случае, если перед остаточными параметрами объявлены обязательные параметры, то функция будет совместима с любой другой функцией, которая совместима с обязательной частью.

ts

```ts
type T0 = (p0: number, ...rest: any[]) => void;
type T1 = (p0: number, p1: string) => void;
type T2 = (p0: string, p1: string) => void;

let v0: T0 = (p0, ...rest) => {};
let v1: T1 = (p0, p1) => {};
let v2: T2 = (p0, p1) => {};

let v3: T0 = v1; // Ok
let v4: T1 = v0; // Ok
let v5: T2 = v0; // Error
let v6: T0 = v2; // Error
```

Следующий, один из неочевидных моментов совместимости функциональных типов, заключается в том, что сигнатура с меньшим числом параметров, совместима с сигнатурой, с большим числом параметров, но не наоборот. Это правило верно при условии соблюдения предыдущих правил, относящихся к совместимости типов.

ts

```ts
type T0 = (p0: number, p1: string) => void;
type T1 = () => void;

let v0: T0 = () => {}; // Ok
let v1: T0 = (p: number) => {}; // Ok
let v3: T1 = (p?: number) => {}; // Ok -> необязательный параметр
let v4: T1 = (p: number) => {}; // Error -> обязательных параметров
больше чем в типе T1
```

На данный момент уже известно, что два объектных типа (`:T = S`), на основании структурной типизации, считаются совместимыми, если в типе `S` присутствуют все признаки типа `T` . Помимо этого, тип `S` может быть более специфичным, чем тип `T` . Простыми словами, тип `S` , помимо всех признаков, присущих в типе `T` , также может обладать признаками которые в типе `T` отсутствуют, но не наоборот. Если ещё более просто, то больший тип совместим с меньшим типом данных. В случае с параметрами функциональных типов, все с точностью наоборот.

ts

```ts
type T = (p0: number) => void;

let v0: T = (p0) => {}; // Ok, такое же количество параметров
let v1: T = () => {}; // Ok, параметров меньше
let v2: T = (p0, p1) => {}; // Error, параметров больше
```

Такое поведение проще всего объяснить на примере работы с методами массива. За основу будет взята декларация метода `forEach` из библиотеки `lib.es5.d.ts` .

ts

```ts
forEach(callbackFn: (value: T, index: number, array: T[]) => void,
thisArg?: any): void;
```

В данном случае нужно обратить внимание на функциональный тип первого параметра, описывающего три других параметра.

ts

```ts
callbackFn: (value: T, index: number, array: T[]) => void;
```

Если бы функциональный тип с большим числом параметров не был совместим с функциональным типом с меньшим числом параметров, то при работе с методом `forEach` , при необходимости только в одном, первом параметре, обязательно бы приходилось создавать *callback* со всеми тремя параметрами, что привело бы к излишнему коду.

ts

```ts
class Animal { name: string; }
class Elephant extends Animal {}
class Lion extends Animal {}

let animals: Animal[] = [
    new Elephant(),
    new Lion()
];

let animalNames: string[] = [];
```

```
animals.forEach((value, index, source) => { // Плохо
    animalNames.push(value.name);
});

animals.forEach(value => { // Хорошо
    animalNames.push(value.name);
});
```

Кроме того, обобщенные функции, чьим параметрам в качестве типа указан параметр типа, проверяются на совместимость по стандартному сценарию.

ts

```ts
function f0<T>(p0: T): void {}
function f1<T, S>(p0: T, p1: S): void {}

type T0 = typeof f0;
type T1 = typeof f1;

let v0: T0 = f1; // Error
let v1: T1 = f0; // Ok
```

Также стоит знать, что параметры, принадлежащие к конкретным типам, совместимы с параметрами, которым в качестве типов указаны параметры типа, но не наоборот.

ts

```ts
function f0<T>(p: T): void {}
function f1(p: number): void {}

type T0 = typeof f0;
type T1 = typeof f1;

let v0: T0 = f1; // Error, параметр типа T не совместим с параметром
типа number
let v1: T1 = f0; // Ok, параметр типа number совместим с параметром
типа T
```

Помимо того, что две сигнатуры считаются совместимыми если участвующие в проверке параметры принадлежат к одному типу, они также считаются совместимыми при совместимости типов этих параметров. Но с этим связана ещё одна неочевидность. Как известно, в контексте объектных типов, если тип `T1` не идентичен полностью типу `T2`, и при этом тип `T1` совместим с типом `T2`, то значит тип `T2` будет совместим с типом `T1` через операцию приведения типов.

ts

```ts
class T0 { f0: number; }
class T1 { f0: number; f1: string; }
let v0: T0 = new T1(); // Ok -> неявное преобразование типов
```

```
let v1: T1 = new T0(); // Error
let v2: T1 = new T0() as T1; // Ok -> явное приведение типов
```

С типами в аннотации параметров функций все, то же самое, только не требуется явного преобразование типов. Такое поведение называется бивариантностью параметров и создано для того, что бы сохранить совместимость с распространенными в *JavaScript* практиками. Подробно бивариантность была рассмотрена в главе “Экскурс в типизацию - Совместимость типов на основе вариантности”.

ts

```
class T0 { f0: number; }
class T1 { f0: number; f1: string; }

function f0(p: T1): void {}
function f1(p: T0): void {}

type FT0 = typeof f0;
type FT1 = typeof f1;

// бивариантное поведение
let v0: FT0 = f1; // Ok, параметр с типом T1 совместим с параметром
принадлежащим к типу T0. Кроме того, тип T1 совместим с типом T0.
let v1: FT1 = f0; // Ok, параметр с типом T0 совместим с параметром
принадлежащем к типу T1. Но тип T0 не совместим с типом T1  без явного
приведения.
```

Изменить поведение бивариантного сопоставления параметров можно с помощью опции компилятора `--strictFunctionTypes` . Установив флаг `--strictFunctionTypes` в `true` , сопоставление будет происходить по контрвариантным правилам (глава “Экскурс в типизацию - Совместимость типов на основе вариантности”).

ts

```
class T0 { f0: number; }
class T1 { f0: number; f1: string; }

function f0(p: T1): void {}
function f1(p: T0): void {}

type FT0 = typeof f0;
type FT1 = typeof f1;

// контрвариантное поведение
let v0: FT0 = f1; // Ok
let v1: FT1 = f0; // Error
```

[41.2] Совместимость возвращаемого значения

Первое, на, что стоит обратить внимание, это ожидаемое поведение при проверке на совместимость возвращаемых типов. Другими словами, две сигнатуры считаются совместимыми, если их типы, указанные в аннотации возвращаемого значения, совместимы по правилам структурной типизации, которой подчиняются все объекты в *TypeScript*.

ts

```ts
class T0 { f0: number; }
class T1 { f0: number; f1: string; }

type FT0 = () => T0;
type FT1 = () => T1;

let v0: FT0 = () => new T1(); // Ok
let v1: FT1 = () => new T0(); // Error
```

Исключением из этого правила составляет примитивный тип данных `void`. Как стало известно из главы посвященной типу данных `void`, в обычном режиме он совместим только с типами `null` и `undefined`, так как они являются его подтипами. При активной рекомендуемой опции `--strictNullChecks`, примитивный тип `void` совместим только с типом `undefined`.

ts

```ts
let v0: void = null; // Ok and Error с включенным флагом
strictNullChecks
let v1: void = undefined; // Ok
```

Но это правило неверно, если тип `void` указан в аннотации возвращаемого из функции значения. В случаях, когда примитивный тип `void` указан в качестве возвращаемого из функции типа, он совместим со всеми типами, без исключения.

ts

```ts
type T = () => void;

let v0: T = () => 0; // Ok
let v1: T = () => ''; // Ok
let v2: T = () => true; // Ok
let v3: T = () => ({}); // Ok
```

Причину по которой поведение типа `void` при указании его в аннотации возвращаемого из функции значения было изменено лучше рассмотреть на примере работы с массивом, а точнее его методом `forEach`.

Предположим есть два массива. Первый массив состоит из элементов принадлежащих к объектному типу у которого определенно поле `name`. Второй массив предназначен для хранения строк и имеет длину равную `0`.

ts

```ts
class Animal {
    name: string;
}

let animals: Animal[] = [
    new Animal(),
    new Animal()
];
```

Задача заключается в получении имен объектов из первого массива с последующим сохранением их во второй массив.

Для этого потребуется определить стрелочную функцию обратного вызова (*callback*). Слева от стрелки будет расположен один параметр `value`, а справа — операция сохранения имени во второй массив с помощью метода `push`. Если обратится к декларации метода массива `forEach`, то можно убедится, что в качестве функции обратного вызова этот метод принимает функцию у которой отсутствует возвращаемое значение.

ts

```ts
forEach(callbackFn: (value: T, index: number, array: T[]) => void,
thisArg?: any): void;
```

Но в нашем случае в теле функции обратного вызова происходит операция добавления элемента в массив. Результатом этой операции является значение длины массива. То есть, метод `push` возвращает значение, принадлежащий к типу `number`, которое в свою очередь возвращается из стрелочной функции обратного вызова, переданного в метод `forEach`, у которого этот параметр задекларирован как функция возвращающая тип `void`, что противоречит возвращенному типу `number`. В данном случае отсутствие

ошибки объясняется совместимостью типа `void`, используемого в функциональных типах, со всеми остальными типами.

ts

```ts
class Animal { name: string; }

let animals: Animal[] = [
    new Animal(),
    new Animal()
];

let animalNameAll: string[] = [];

animalNameAll.forEach( animal => animalNameAll.push(
animal.name ) ); // forEach ожидает () => void, а получает () =>
number, так как стрелочная функция без тела неявно возвращает значение,
возвращаемое методом push.
```

И напоследок стоит упомянуть, что две обобщенные функции считаются совместимы, если у них в аннотации возвращаемого значения указан параметр типа.

ts

```ts
function f0<T>(p: T): T { return p; }
function f1<S>(p: S): S { return p; }

type T0 = typeof f0;
type T1 = typeof f1;

let v0: T0 = f1; // Ok
let v1: T1 = f0; // Ok
```

Кроме того, параметр типа совместим с любым конкретным типом данных, но не наоборот.

ts

```ts
function f0<T>(p: T): T { return p; }
function f1(p: number): number { return p; }

type T0 = typeof f0;
type T1 = typeof f1;

let v0: T0 = f1; // Error
let v1: T1 = f0; // Ok
```

Глава 42

Совместимость объединений (Union Types)

Поскольку понимание поведения типа `Union` при проверке на совместимость может вызывать противоречия, эта совсем крохотная глава, будет посвящена этому механизму.

[42.0] Совместимость

Чтобы было проще понять суть противоречий возникающих у разработчиков при понимании механизма совместимости типов объединение, стоит начать с повторения совместимости объектных типов.

Как известно к данному моменту, объектный тип `A` совместим с объектным типом `B`, если первый содержит все обязательные признаки второго. Кроме того, члены участвующие в проверке на совместимость не обязаны принадлежать к идентичным типам, достаточно, что бы они также были совместимы. Утрированно всё сказанное можно перефразировать как: *"Тип, обладающий большим количеством совместимых признаков, совместим с типом, обладающим меньшим количеством признаков"*. Или даже *больший тип совместим с меньшим типом*.

ts

```ts
interface Smaller {
    a: number;
    b: string;
```

```
}
interface Bigger {
    a: number;
    b: string;
    c: boolean;
}


declare let small: Smaller;
declare let big: Bigger;

let s: Smaller = big; // Ok
let b: Bigger = small; // Error
```

Любому разработчику начавшему свою карьеру с языка реализующего *ооп* парадигму, подобное поведение кажется само собой разумеющимся. Так вот, с типом объединение (`Union`) все в точности наоборот. Точнее может показаться, что наоборот, хотя на самом деле это совершенно другой случай.

ts

```
type Smaller = number | string;
type Bigger = number | string | boolean;

declare let small: Smaller;
declare let big: Bigger;

let s: Smaller = big; // Error
let b: Bigger = small; // Ok
```

В случае с совместимостью объекта, значение принадлежащие к *большему типу* обладает всеми необходимыми признаками требующихся для успешного выполнения операций предназначенных для *меньшего типа*. В случае с типом объединением чем больше типов его определяют, тем больше шансов, что значение будет принадлежать к типу отсутствующему в *меньшем типе*.

В нашем примере переменная с типом `Bigger` помимо прочего может быть ассоциирована со значением принадлежащим к типу `boolean` , который не определяет множество типа `Smaller` .

ts

```
type Smaller = number | string;
type Bigger = number | string | boolean;


declare let small: Smaller;
declare let big: Bigger;

/**[0] */
let s: Smaller = big; // Error
```

```
/**[1] */
let b: Bigger = small; // Ok

/**
 * [0] переменная big может иметь значение принадлежащие
 * к типу boolean которое отсутствует в типе Smaller. Поэтому
 * переменная big не может быть присвоена переменной с Smaller.
 *
 * [1] И наоборот. Поскольку переменная small может иметь значение
 * принадлежащие либо к number либо string, её можно присвоить
переменной
 * с типом Bigger поскольку множество определяющее его включает данные
типы.
 */
```

Глава 43

Типизация в TypeScript

Данная глава поможет разработчикам не просто писать типизированный код, а делать это в полной мере осмысленно. Для этого необходимо ещё раз повторить все концепции нашедшие свое применение в языке *TypeScript*.

[43.0] Общие сведения

Самое время взять паузу и рассмотреть типизацию в *TypeScript* более детально через призму полученных знаний.

Итак, что известно о *TypeScript*? *TypeScript* это язык:

1. Статически типизированный с возможностью динамического связывания
2. Сильно типизированный
3. Явно типизированный с возможностью вывода типов
4. Совместимость типов в *TypeScript* проходит по правилам структурной типизации
5. Совместимость типов зависит от вариантности, чей конкретный вид определяется конкретным случаем

Кроме этого, существуют понятия являющиеся частью перечисленных, но в *TypeScript*, выделенные в отдельные определения. По этой причине они будут рассматриваться отдельно. Такими понятиями являются:

1. Наилучший общий тип
2. Контекстный тип

Начнем с повторения определений в том порядке, в котором они были перечислены.

[43.1] Статическая типизация (static typing)

Статическая типизация обуславливается тем, что связывание с типом данных происходит на этапе компиляции и при этом тип не может изменится на протяжении всего своего существования.

Статическая типизация в *TypeScript* проявляется в том, что к моменту окончания компиляции компилятору известно к какому конкретному типу принадлежат конструкции нуждающиеся в аннотации типа.

[43.2] Сильная типизация (strongly typed)

Язык с *сильной типизацией* не позволяет операции с несовместимыми типами, а также не выполняет явного преобразования типов.

Сильная типизация в *TypeScript* проявляет себя в случаях схожих с операцией сложения числа с массивом. В этом случае компилятор выбрасывает ошибки.

ts

```ts
const value = 5 + []; // Error
```

[43.3] Явно типизированный (explicit typing) с выводом типов (type inference)

Язык с *явной типизацией* предполагает, что указание типов будет выполнено разработчиком. Но современные языки с явной типизацией имеют возможность указывать типы неявно. Это становится возможным за счет механизма *вывода типов*.

Вывод типов — это возможность компилятора (интерпретатора) самостоятельно выводить-указывать тип данных на основе анализа выражения.

В *TypeScript*, если тип не указывается явно, компилятор с помощью вывода типов выводит и указывает тип самостоятельно.

ts

```ts
var animal: Animal = new Animal(); // animal: Animal
var animal = new Animal(); // animal: Animal
```

[43.4] Совместимость типов (Type Compatibility), структурная типизация (structural typing)

Совместимость типов — это механизм по которому происходит сравнение типов.

Простыми словами, совместимость типов — это совокупность правил, на основе которых программа анализируя два типа данных, выясняет, производить над ними операции считая их совместимыми, либо для этого требуется преобразование. Правила совместимости типов делятся на три вида, один из которых имеет название структурная типизация.

Структурная Типизация — это принцип определяющий совместимость типов основываясь не на иерархии наследования или явной реализации интерфейсов, а на их описании.

Несмотря на то, что `Bird` и `Fish` не имеют явно заданного общего предка, *TypeScript* разрешает присваивать экземпляр класса `Fish` переменной с типом `Bird` (и наоборот).

ts

```ts
class Bird { name; }
class Fish { name; }

var bird: Bird = new Fish();
var fish: Fish = new Bird();
```

В таких языках, как *Java* или *C#*, подобное поведение недопустимо. В *TypeScript* это становится возможно из-за структурной типизации.

Так как совместимость типов происходит на основе их описания, в первом случае компилятор запоминает все члены типа `Fish`, и если он находит аналогичные члены в типе `Bird`, то они считаются совместимыми. То же самое компилятор проделывает тогда, когда во втором случае присваивает экземпляр класса `Bird` переменной с типом `Fish`. Так как оба типа имеют по одному полю, с одинаковым типом и идентификатором, то они считаются совместимыми.

Если добавить классу `Bird` поле `wings`, то при попытке присвоить его экземпляр переменной с типом `Fish` возникнет ошибка, так как в типе `Fish` отсутствует после `wings`. Обратное действие, то есть присвоение экземпляра класса `Bird` переменной с типом `Fish`, ошибки не вызовет, так как в типе `Bird` будут найдены все члены объявленные в типе `Fish`.

ts

```ts
class Bird { name; wings; }
class Fish { name; }

var bird: Bird = new Fish(); // Error
var fish: Fish = new Bird();
```

Стоит добавить, что правилам структурной типизации подчиняются все объекты в *TypeScript*. А, как известно, в *JavaScript* все, кроме примитивных типов, объекты. Это же утверждение верно и для *TypeScript*.

[43.5] Вариантность (variance)

Простыми словами, *вариантность* — это механизм, определяющий правила на основе которых принимается решение о совместимости двух типов. Правила зависят от конкретного вида вариантности — *ковариантность*, *контравариантность*, *бивариантность* и *инвариантность*. В случае с *TypeScript* нас интересуют первые три.

Ковариантность позволяет большему типу быть совместимым с меньшим типом.

ts

```ts
interface IAnimal {
    type: string;
}

interface IBird extends IAnimal {
    fly(): void;
}

function f0(): IAnimal {
    const v: IAnimal = {
        type: 'animal'
    };

    return v;
}

function f1(): IBird {
    const v: IBird = {
        type: 'bird',
        fly() {

        }
    };

    return v;
}


type T0 = typeof f0;
type T1 = typeof f1;


let v0: T0 = f1; // Ok
let v1: T1 = f0; // Error
```

Контравариантность позволяет меньшему типу быть совместимым с большим типом.

ts

```ts
interface IAnimal {
    type: string;
}

interface IBird extends IAnimal {
    fly(): void;
}

function f0(p: IAnimal): void {}
function f1(p: IBird): void {}

type T0 = typeof f0;
type T1 = typeof f1;
```

```
let v0: T0 = f1; // Error
let v1: T1 = f0; // Ok
```

Бивариантность, доступная исключительно для параметров функций при условии, что флаг `--strictFunctionTypes` установлен в значение `false`, делает возможной совместимость как большего типа с меньшим, так и наоборот — меньшего с большим.

ts

```
interface IAnimal {
    type: string;
}

interface IBird extends IAnimal {
    fly(): void;
}

function f0(p: IAnimal): void {}
function f1(p: IBird): void {}

type T0 = typeof f0;
type T1 = typeof f1;

let v0: T0 = f1; // Ok, (--strictFunctionTypes === false)
let v1: T1 = f0; // Ok
```

Не будет лишним упомянуть, что бивариантность снижает уровень типобезопасности программы и поэтому рекомендуется вести разработку с флагом `--strictFunctionTypes` установленным в значение `true`.

[43.6] Наилучший общий тип (Best common type)

С выводом типов в *TypeScript* связано такое понятие, как наилучший общий тип. Это очень простое правило, название которого в большей мере раскрывает его суть.

Как уже было сказано, *TypeScript* — статически типизированный язык, и поэтому он пытается всему указать типы. В случаях, когда тип не был указан явно, в работу включается вывод типов. Предположим, что существует массив ссылка на который присваивается переменной объявленной без явного указания типа. Для того, что бы вывод типов смог вывести тип для переменной, ему нужно проанализировать данные хранящиеся в массиве (если они хранятся).

Для примера представьте массив хранящий экземпляры классов `Animal`, `Elephant` и `Lion`, последние два из которых расширяют первый. И, кроме того, ссылка на данный массив присваивается переменной.

ts

```ts
class Animal {}
class Elephant extends Animal {}
class Lion extends Animal {}

const animalAll = [
    new Elephant(),
    new Lion(),
    new Animal()
]; // animalAll: Animal[]
```

Так как *TypeScript* проверяет совместимость типов по правилам структурной типизации и все три типа идентичны с точки зрения их описания, то с точки зрения вывода типов все они идентичны. Поэтому он выберет в качестве типа тот который является более общим, то есть тип `Animal`.

Если типу `Elephant` будет добавлено поле, например, хобот (`trunk`), что сделает его отличным от всех, то вывод типов будет вынужден указать массиву базовый для всех типов тип `Animal`.

ts

```ts
class Animal {}
class Elephant extends Animal { trunk; }
class Lion extends Animal {}

const animalAll = [
    new Elephant(),
    new Lion (),
    new Animal()
]; // animalAll: Animal[]
```

В случае, если в массиве не будет присутствовать базовый для всех типов тип `Animal`, то вывод типов будет расценивать массив как принадлежащий к типу объединение `Elephant | Lion`.

ts

```
class Animal {}
class Elephant extends Animal { trunk; }
class Lion extends Animal {}

let animalAll = [
    new Elephant(),
    new Lion()
]; // animalAll: (Elephant | Lion)[]
```

Как видно, ничего неожиданного или сложного в теме наилучшего общего типа совершенно нет.

[43.7] Контекстный тип (Contextual Type)

Контекстным называется тип, который при не явном объявлении указывается за счет декларации контекста, а не с помощью вывода типов.

Лучшим примером контекстного типа может служить подписка `document` на событие мыши `mousedown`. Так как у слушателя события тип параметра `event` не указан явно, а также ему в момент объявления не было присвоено значение, то вывод типов должен был указать тип `any`. Но в данном случае компилятор указывает тип `MouseEvent`, потому, что именно он указан в декларации типа слушателя событий. В случае подписания `document` на событие `keydown`, компилятор указывает тип как `KeyboardEvent`.

ts

```
document.addEventListener('mousedown', (event) => { }); // event:
MouseEvent
document.addEventListener('keydown', (event) => { }); // event:
KeyboardEvent
```

Для того, что бы понять, как это работает, опишем случай из жизни зоопарка — представление с морским львом. Для этого создадим класс морской лев `SeaLion` и объявим в нем два метода: вращаться (`rotate`) и голос (`voice`).

ts

```
class SeaLion {
    rotate(): void { }
    voice(): void { }
}
```

Далее, создадим класс дрессировщик `Trainer` и объявим в нем метод `addEventListener` с двумя параметрами: `type` с типом `string` и `handler` с типом `Function`.

ts

```
class Trainer {
    addEventListener(type: string, handler: Function) {}
}
```

Затем объявим два класса события, выражающие команды дрессировщика `RotateTrainerEvent` и `VoiceTrainerEvent`.

ts

```
class RotateTrainerEvent {}
class VoiceTrainerEvent {}
```

После объявим два псевдонима (`type`) для литеральных типов `string`. Первому зададим имя `RotateEventType` и в качестве значения присвоим строковой литерал `"rotate"`. Второму зададим имя `VoiceEventType` и в качестве значения присвоим строковой литерал `"voice"`.

ts

```
type RotateEventType = "rotate";
type VoiceEventType = "voice";
```

Теперь осталось только задекларировать ещё два псевдонима типов для функциональных типов у обоих из которых будет один параметр `event` и отсутствовать возвращаемое значение. Первому псевдониму зададим имя `RotateTrainerHandler`, а его параметру установим тип `RotateTrainerEvent`. Второму псевдониму зададим имя `VoiceTrainerHandler`, а его параметру установим тип `VoiceTrainerEvent`.

ts

```
type RotateTrainerHandler = (event: RotateTrainerEvent) => void;
type VoiceTrainerHandler = (event: VoiceTrainerEvent) => void;
```

Соберём части воедино. Для этого в классе дрессировщик `Trainer` перегрузим метод `addEventListener`. У первого перегруженного метода параметр `type` будет иметь тип `RotateEventType`, а параметру `handler` укажем тип `RotateTrainerHandler`. Второму перегруженному методу в качестве типа параметра

`type` укажем `VoiceEventType`, а параметру `handler` укажем тип `VoiceTrainerHandler`.

ts

```
class Trainer {
  addEventListener(type: RotateEventType, handler:
RotateTrainerHandler);
  addEventListener(type: VoiceEventType, handler: VoiceTrainerHandler);
  addEventListener(type: string, handler: Function) {}
}
```

Осталось только убедиться, что все работает правильно. Для этого создадим экземпляр класса `Trainer` и подпишемся на события. Сразу можно увидеть подтверждение того, что цель достигнута. У слушателя события `RotateTrainerEvent` параметру `event` указан контекстный тип `RotateTrainerEvent`. А слушателю события `VoiceTrainerEvent` параметру `event` указан контекстный тип `VoiceTrainerEvent`.

ts

```
type RotateTrainerHandler = (event: RotateTrainerEvent) => void;
type VoiceTrainerHandler = (event: VoiceTrainerEvent) => void;

type RotateEventType = "rotate";
type VoiceEventType = "voice";

class RotateTrainerEvent {}
class VoiceTrainerEvent {}

class SeaLion {
    rotate(): void {}
    voice(): void {}
}

class Trainer {
    addEventListener(type: RotateEventType, handler:
RotateTrainerHandler);
    addEventListener(type: VoiceEventType, handler:
VoiceTrainerHandler);
    addEventListener(type: string, handler: Function) {}
}

let seaLion: SeaLion = new SeaLion();

let trainer: Trainer = new Trainer();
trainer.addEventListener('rotate', (event) => seaLion.rotate());
trainer.addEventListener('voice', (event) => seaLion.voice());
```

Глава 44

Оператор keyof, Lookup Types, Mapped Types, Mapped Types - префиксы + и -

Для того, что бы повысить уровень выявления ошибок и при этом сократить время разработки программы, создатели *TypeScript* не прекращают радовать разработчиков добавлением новых возможностей для взаимодействия с типами данных. Благодаря усилиям разработчиков со всего земного шара, стало осуществимо получать тип объединение, полученный на основе как ключей конкретного типа, так и ассоциированных с ними типами. Кроме этого, возможность определять новый тип в процессе итерации другого типа, используя при этом различные выражения, превращает типизированный мир в фантастическую страну. Единственное, что требуется от разработчика, понимание этих процессов, которым и будет посвящена текущая глава.

[44.0] Запрос ключей keyof

В *TypeScript* существует возможность выводить все публичные, не статические, принадлежащие типу ключи и на их основе создавать литеральный объединенный тип (`Union`). Для получения ключей нужно указать оператор `keyof` , после которого указывается тип, чьи ключи будут объединены в тип объединение `keyof Type` .

Оператор `keyof` может применяться к любому типу данных.

ts

```ts
type AliasType = { f1: number, f2: string };

interface IInterfaceType {
  f1: number;
  f2: string;
}

class ClassType {
  f1: number;
  f2: string;
}

let v1: keyof AliasType; // v1: "f1" | "f2"
let v2: keyof IInterfaceType; // v2: "f1" | "f2"
let v3: keyof ClassType; // v3: "f1" | "f2"
let v4: keyof number; // v4: "toString" | "toFixed" | "toExponential" |
"toPrecision" | "valueOf" | "toLocaleString"
```

Как уже было замечено, оператор `keyof` выводит только публичные не статические ключи типа.

ts

```ts
class Type {
    public static fieldClass: number;
    public static methodClass(): void {}

    private privateField: number;
    protected protectedField: string;
    public publicField: boolean;

    public constructor() {}

    public get property(): number { return NaN; }
    public set property(value: number) {}
    public instanceMethod(): void {}
}

let v1: keyof Type; // a: "publicField" | "property" | "instanceMethod"
```

В случае, если тип данных не содержит публичных ключей, оператор `keyof` выведет тип `never`.

ts

```ts
type AliasType = {};

interface IInterfaceType {}

class ClassType {
```

```
    private f1: number;
    protected f2: string;
}

let v1: keyof AliasType; // v1: never
let v2: keyof IInterfaceType; // v2: never
let v3: keyof ClassType; // v3: never
let v4: keyof object; // v4: never
```

Оператор `keyof` также может использоваться в объявлении обобщенного типа данных. Точнее, с помощью оператора `keyof` можно получить тип, а затем расширить его параметром типа. Важно понимать, что в качестве значения по умолчанию может выступать только тип, совместимый с объединенным типом, полученным на основе ключей.

ts

```
function f1<T, U extends keyof T = keyof T>(): void {}
```

Напоследок стоит упомянуть об одном не очевидном моменте: оператор `keyof` можно совмещать с оператором `typeof` (*Type Queries*).

ts

```
class Animal {
    public name: string;
    public age: number;
}

let animal = new Animal();

let type: typeof animal; // type: { name: string; age: number; }
let union: keyof typeof animal; // union: "name" | "age"
```

[44.1] Поиск типов (Lookup Types)

Если оператор `keyof` выбирает все доступные ключи, то с помощью поиска типов можно получить заданные типы по известным ключам. Получить связанный с ключом тип можно с помощью скобочной нотации, в которой через оператор вертикальная черта `|` будут перечислены от одного и более ключа, существующего в типе. В качестве типа данных могут выступать только интерфейсы, классы и в ограниченных случаях операторы типа.

В случаях, когда в качестве типа данных выступает интерфейс, то получить можно все типы, без исключения. При попытке получить тип несуществующего ключа возникнет ошибка.

ts

```ts
interface IInterfaceType {
    p1: number;
    p2: string;
}

let v1: IInterfaceType['p1']; // v1: number
let v2: IInterfaceType['p2']; // v2: string
let union: IInterfaceType['p1' | 'p2']; // union: number | string
let notExist: IInterfaceType['notExist']; // Error -> Property
'notExist' does not exist on type 'IAnimal'
```

Если в качестве типа выступает класс, то получить типы можно только у членов его экземпляра. При попытке получить тип несуществующего члена возникнет ошибка.

ts

```ts
class ClassType {
    public static publicFieldClass: number;

    public publicInstanceField: number;
    protected protectedInstanceField: string;
    private privateInstanceField: boolean;

    public get propertyInstance(): number { return NaN; }
    public set propertyInstance(value: number) {}

    public methodInstance(): void {}
}

let publicFieldClass: ClassType['publicFieldClass']; // Error

let publicFieldInstance: ClassType['publicInstanceField']; //
publicFieldInstance: number
let protectedFieldInstance: ClassType['protectedInstanceField']; //
protectedFieldInstance: string
let privateFieldInstance: ClassType['privateInstanceField']; //
privateFieldInstance: boolean
let propertyInstance: ClassType['propertyInstance']; //
propertyInstance: number
let methodInstance: ClassType['methodInstance']; // methodInstance: ()
=> void

let notExist: ClassType['notExist']; // Error
```

Нельзя переоценить вклад возможностей поиска типов, которые пришлись на динамическую часть типизированного мира *TypeScript*. Благодаря поиску типов в паре с оператором `keyof` появилась возможность, позволяющая выводу типов устанавливать связь между динамическими ключами и их типами. Это в свою очередь позволяет производить дополнительные проверки, которые повышают типобезопасность кода.

ts

```
class Model<T> {
    constructor(private entity: T) {}

    public getValueByName<U extends keyof T>(key: U): T[U] {
        return this.entity[key];
    }
}

interface IAnimalModel {
    id: string;
    age: number;
}

let json = '"{"id": "animal", "age": 0}"';
let entity: IAnimalModel = JSON.parse(json);

let userModel: Model<IAnimalModel> = new Model(entity);

let id = userModel.getValueByName('id'); // id: string
let age = userModel.getValueByName('age'); // age: number
```

[44.2] Сопоставление типов (Mapped Types)

Сопоставленные типы — это типы данных, которые при помощи механизма итерации модифицируют лежащие в основе конкретные типы данных.

В *TypeScript* существует возможность определения типа, источником ключей которого выступает множество определяемое литеральными строковыми типами. Подобные типы обозначаются как *сопоставленные типы* (`Mapped Types`) и определяются исключительно на основе псевдонимов типов (`Type Alias`), объявление которых осуществляется при помощи ключевого слова `type` . Тело сопоставимого типа, заключенное в фигурные скобки `{}` , включает в себя одно единственное выражение, состоящие из двух частей разделенных двоеточием.

ts

```
type СопоставимыйТип = {
    ЛеваяЧастьВыражения: ПраваяЧастьВыражения;
}
```

В левой части выражения располагается обрамленное в квадратные скобки `[]` выражение, предназначенное для работы с множеством, а в правой части определяется произвольный тип данных.

ts

```
type СопоставимыйТип = {
    [ВыражениеДляРаботыСМножеством]: ПроизвольныйТипДанных;
}
```

Выражение описывающее итерацию элементов представляющих ключи, также состоит из двух частей, разделяемых оператором `in` (`[ЛевыйОперанд in ПравыйОперанд]`). В качестве левого операнда указывается произвольный идентификатор, который в процессе итерации, последовательно будет ассоциирован со значениями множества указанного в правой части (`[ПроизвольныйИдентификатор in Множество]`).

ts

```
type СопоставимыйТип = {
    [ПроизвольныйИдентификатор in Множество]: ПроизвольныйТипДанных;
}
```

Как уже было сказано, в роли идентификатора может выступать любой идентификатор.

ts

```
type СопоставимыйТип = {
    [Key in Множество]: ПроизвольныйТипДанных;
}

// или

type СопоставимыйТип = {
    [K in Множество]: ПроизвольныйТипДанных;
}
```

Множество может быть определенно как единственным литеральным строковым типом (`ElementLiteralStringType`), так и его множеством, составляющим тип объединение (`Union Type`) (`FirstElementLiteralStringType | SecondElementLiteralStringType`).

ts

```
// множество с одним элементом
type СопоставимыйТип = {
    [K in "FirstLiteralStringType"]: ПроизвольныйТипДанных;
}

// или

// множество с несколькими элементами
type СопоставимыйТип = {
    [K in "FirstLiteralStringType" | "SecondLiteralStringType"] :
ПроизвольныйТипДанных;
}

// или

type LiteralStringType = "FirstLiteralStringType" |
"SecondLiteralStringType";

// множество с несколькими элементами вынесенных в тип Union
type СопоставимыйТип = {
    [K in LiteralStringType]: ПроизвольныйТипДанных;
}
```

Результатом определения сопоставленного типа является новый объектный тип, состоящий из ключей ассоциированных с произвольным типом.

ts

```
type ABC = "a" | "b" | "c";

type ABCWithString = {
    [K in ABC]: string;
}

// или

type ABCWithNumber = {
    [K in ABC]: number;
}

declare function abcWithString(params: ABCWithString): void;

abcWithString({a: '', b: '', c: ''}); // Ok
abcWithString({}); // Error, missing properties 'a', 'b', 'c'
abcWithString({a: '', b: ''}); // Error, missing property 'c'
abcWithString({a: '', b: '', c: 5}); // Error, type number is not type
string

declare function abcWithNumber(params: ABCWithNumber): void;

abcWithNumber({a: 0, b: 0, c: 0}); // Ok
abcWithNumber({}); // Error, missing properties 'a', 'b', 'c'
```

```
abcWithNumber({a: 0, b: 0}); // Error, missing property 'c'
abcWithNumber({a: 0, b: 0, c: ''}); // Error, type string is not type
number
```

От статического указания итерируемого типа мало пользы, поэтому `Mapped Types` лучше всего раскрывают свой потенциал при совместной работе с известными к этому моменту запросом ключей (`keyof`) и поиском типов (`Lookup Types`), оперирующих параметрами типа (`Generics`).

ts

```
type MappedType<T> = {
    [K in keyof T]: T[K];
}

// или

type MappedType<T, U extends keyof T> = {
    [K in U]: T[K];
}
```

В первом случае в выражении `[P in keyof T]: T[P];` первым действием выполняется вычисление оператора `keyof` над параметром типа `T` . В его результате ключи произвольного типа преобразуются во множество, то есть в тип `Union` , элементы которого принадлежат к литеральному строковому типу данных. Простыми словами операция `keyof T` заменяется на только, что полученный тип `Union` `[P in Union]: T[P];` , над которым на следующим действии выполняется итерация.

Во втором случае `MappedType<T, U extends keyof T>` оператор `keyof` также преобразует параметр типа `T` в тип `Union` , который затем расширяет параметр типа `U` , тем самым получая все его признаки, необходимые для итерации в выражении `[K in U]` .

С полученным в итерации `[K in U]` ключом `K` ассоциируется тип ассоциированный с ним в исходном типе и полученный с помощью механизма поиска типов `T[K]` .

Совокупность описанных механизмов позволяет определять не только новый тип, но и создавать модифицирующие типы, которые будут способны добавлять модификаторы, как например `readonly` или `?:` .

ts

```
type ReadonlyMember<T> = {
    readonly [P in keyof T]: T[P];
}

interface IAnimal {
    name: string;
    age: number;
}
```

```
let animal: ReadonlyMember<IAnimal>;  // animal: { readonly name:
string; readonly age: number; }
```

Как уже было замечено, в правой части выражения можно указать любой тип данных, в том числе и объединение, включающего тип полученный при помощи механизма поиска типов.

ts

```ts
type Nullable<T> = {
    [P in keyof T]: T[P] | null;
}

type Stringify<T> = {
    [P in keyof T]: string;
}

interface IAnimal {
    name: string;
    age: number;
}

let nullable: Nullable<IAnimal>; // { name: string | null; age: number
| null; }
let stringify: Stringify<IAnimal>; // { name: string; age: string; }
```

Сопоставленные типы не могут содержать более одной итерации в типе, а также не могут содержать объявление других членов.

ts

```ts
type AliasType<T, U> = {
    [P in keyof T]: T[P]; // Ok
    [V in keyof U]: U[V]; // Error
    f1: number; // Error
}
```

К тому же в *TypeScript* существует несколько готовых типов, таких как `Readonly<T>`, `Partial<T>`, `Record<K, T>` и `Pick<T, K>` (глава “Расширенные типы - Readonly, Partial, Required, Pick, Record”).

Кроме того, сопоставленные типы вместе с *шаблонными литеральными строковыми типами* способны переопределить исходные ключи при помощи ключевого слова `as` указываемого после строкового перечисления.

ts

```
type T = {
    [K in STRING_VALUES as NEW_KEY]: K // K преобразованный
}
```

Таким образом совмещая данный механизм с *шаблонными литеральными строковыми типами* можно добиться переопределения исходных ключей.

ts

```
type ToGetter<T> = `get${capitalize T}`;
type Getters<T> = {
    [K in keyof T as ToGetter<K>]: () => T[K];
}

type Person = {
    name: string;
    age: number;
}

/**
 * type T = {
 *  getName: () => string;
 *  getAge: () => number;
 * }
 */
type T = Getters<Person>
```

[44.3] Префиксы + и - в сопоставленных типах

Сопоставленные типы позволяют добавлять модификаторы, но не позволяют их удалять, что в свою очередь имеет большое значение в случае с гомоморфными типами, которые по умолчанию сохраняют модификаторы своего базового типа (гомоморфные типы будут рассмотрены в главе “Расширенные типы - Readonly, Partial, Required, Pick, Record”).

Для разрешения данной проблемы, к модификаторам в типах сопоставления, были добавлены префиксы `+` и `-`, с помощью которых реализуется поведение модификатора — добавить (`+`) или удалить (`-`).

ts

```
type AddModifier<T> = {
    +readonly [P in keyof T]+?: T[P]; // добавит модификаторы readonly
```

```
и ? (optional)
};
type RemoveModifier<T> = {
    -readonly [P in keyof T]-?: T[P]; // удалит модификаторы readonly
и ? (optional)
};

interface IWithoutModifier { field: string; }
interface IWithModifier { readonly field?: string; }

/**
 * Добавление модификаторов
 * было { field: string; }
 * стало { readonly field?: string; }
 */
let addingModifier: AddModifier<IWithoutModifier> = { field: '' };
let withoutModifier: IWithoutModifier = { field: '' };

addingModifier.field = ''; // Error
withoutModifier.field = ''; // Ok

/**
 * Удаление модификаторов
 * было { readonly field?: string; }
 * стало { field: string; }
 */
let removingModifier: RemoveModifier<IWithModifier> = { field: '' };
let withModifier: IWithModifier = { field: '' };

removingModifier.field = ''; // Ok
withModifier.field = ''; // Error
```

Глава 45

Условные типы (Conditional Types)

Определение динамических типов при помощи условных выражений, очень важный момент описания сложных типизированных сценариев, на примере которых, в данной главе будет рассмотрен механизм определения *условных типов*.

[45.0] Условные типы на практике

Условные типы (Conditional Types) — это типы, способные принимать одно из двух значений, основываясь на принадлежности одного типу к другому. Условные типы семантически схожи с тернарным оператором.

ts

```ts
T extends U ? T1 : T2
```

В блоке выражения с помощью ключевого слова `extends` устанавливается принадлежность к заданному типу. Если тип, указанный слева от ключевого слова `extends`, совместим с типом, указанным по правую сторону, то условный тип будет принадлежать к типу `T1`, иначе — к типу `T2`. Стоит заметить, что в качестве типов `T1` и `T2` могут выступать, в том числе и другие условные типы, что в свою очередь создаст цепочку условий определения типа.

Помимо того, что невозможно переоценить пользу от условных типов, очень сложно придумать минимальный пример, который бы эту пользу проиллюстрировал. Поэтому в

этой главе будут приведены лишь бессмысленные примеры, демонстрирующие принцип их работы.

ts

```ts
type T0<T> = T extends number ? string : boolean;

let v0: T0<5>; // let v0: string
let v1: T0<'text'>; // let v1: boolean

type T1<T> = T extends number | string ? object : never;

let v2: T1<5>; // let v2: object
let v3: T1<'text'>; // let v3: object
let v4: T1<true>; // let v2: never

type T2<T> = T extends number ? "Ok" : "Oops";

let v5: T2<5>; // let v5: "Ok"
let v6: T2<'text'>; // let v6: "oops"

// вложенные условные типы

type T3<T> =
    T extends number ? "IsNumber" :
    T extends string ? "IsString" :
    "Oops";

let v7: T3<5>; // let v7: "IsNumber"
let v8: T3<'text'>; // let v8: "IsString"
let v9: T3<true>; // let v9: "Oops"
```

Нужно быть внимательным, когда в условиях вложенных условных типов проверяются совместимые типы, так как порядок условий может повлиять на результат.

ts

```ts
type T0<T> =
    T extends IAnimal ? "animal" :
    T extends IBird ? "bird" :
    T extends IRaven ? "raven" :
    "no animal";

type T1<T> =
    T extends IRaven ? "raven" :
    T extends IBird ? "bird" :
    T extends IAnimal ? "animal" :
    "no animal";

// всегда "animal"
let v0:T0<IAnimal>; // let v0: "animal"
let v1: T0<IBird>; // let v1: "animal"
```

```
let v2: T0<IRaven>; // let v2: "animal"

// никогда "bird"
let v3:T1<IRaven>; // let v3: "raven"
let v4: T1<IBird>; // let v4: "raven"
let v5: T1<IAnimal>; // let v5: "animal"
```

Если в качестве аргумента условного типа выступает тип объединение (`Union` , глава "Типы - Union, Intersection"), то условия будут выполняться для каждого типа, составляющего объединенный тип.

ts

```
interface IAnimal { type: string; }
interface IBird extends IAnimal { fly(): void; }
interface IRaven extends IBird {}

type T0<T> =
    T extends IAnimal ? "animal" :
    T extends IBird ? "bird" :
    T extends IRaven ? "raven" :
    "no animal";

type T1<T> =
    T extends IRaven ? "raven" :
    T extends IBird ? "bird" :
    T extends IAnimal ? "animal" :
    "no animal";

// всегда "animal"
let v0:T0<IAnimal | IBird>; // let v0: "animal"
let v1: T0<IBird>; // let v1: "animal"
let v2: T0<IRaven>; // let v2: "animal"

// никогда "bird"
let v3:T1<IAnimal | IRaven>; // let v3: "raven"
let v4: T1<IBird>; // let v4: "raven"
let v5: T1<IAnimal | IBird>; // let v5: "animal"
```

Помимо конкретного типа, в качестве правого (от ключевого слова `extends`) операнда также может выступать другой параметр типа.

ts

```
type T0<T, U> = T extends U ? "Ok" : "Oops";

let v0: T0<number, any>; // Ok
let v1:T0<number, string>; // Oops
```

[45.1] Распределительные условные типы (Distributive Conditional Types)

Условные типы, которым в качестве аргумента типа устанавливается объединенный тип (`Union Type` , глава "Типы - Union, Intersection"), называются *распределительные условные типы* (`Distributive Conditional Types`). Называются они так, потому, что каждый тип, составляющий объединенный тип, будет распределен таким образом, что бы выражение условного типа было выполнено для каждого. Это, в свою очередь может определить условный тип, как тип объединение.

ts

```ts
type T0<T> =
    T extends number ? "numeric" :
    T extends string ? "text" :
    "other";

let v0: T0<string | number>; // let v0: "numeric" | "text"
let v1: T0<string | boolean>; // let v1: "text" | "other"
```

Для лучшего понимания процесса происходящего при определении условного типа в случае, когда аргумент типа принадлежит к объединению, стоит рассмотреть следующий минимальный пример, в котором будет проиллюстрирован условный тип так, как его видит компилятор.

ts

```ts
// так видит разработчик

type T0<T> =
    T extends number ? "numeric" :
    T extends string ? "text" :
    "other";

let v0: T0<string | number>; // let v0: "numeric" | "text"
let v1: T0<string | boolean>; // let v1: "text" | "other"

// так видит компилятор

type T0<T> =
  // получаем первый тип, составляющий union тип (в данном случае
number) и начинаем подставлять его на место T
```

```
  number extends number ? "numeric" : // number соответствует number? Да! Определяем "numeric"
  T extends string ? "text" :
  "other"

  | // закончили определять один тип, приступаем к другому, в данном случае string

  string extends number ? "numeric" : // string соответствует number? Нет! Продолжаем.
  string extends string ? "text" : // string соответствует string? Да! Определяем "text".
  "other"

  // Итого: условный тип T0<string | number> определен, как "numeric" | "text"
```

[45.2] Вывод типов в условном типе

Условные типы позволяют в блоке выражения объявлять переменные, тип которых будет устанавливать вывод типов. Переменная типа объявляется с помощью ключевого слова `infer` и, как уже говорилось, может быть объявлена исключительно в типе, указанном в блоке выражения расположенном правее оператора `extends`.

Это очень простой механизм, который проще сразу рассмотреть на примере.

Предположим, что нужно установить, к какому типу принадлежит единственный параметр функции.

ts

```ts
function f(param: string): void {}
```

Для этого нужно создать условный тип, в условии которого происходит проверка на принадлежность к типу-функции. Кроме того, аннотация типа единственного параметра этой функции, вместо конкретного типа, будет содержать объявление переменной типа.

ts

```ts
type ParamType<T> = T extends (p: infer U) => void ? U : undefined;

function f0(param: number): void {}
function f1(param: string): void {}
function f2(): void {}
```

```
function f3(p0: number, p1: string): void {}
function f4(param: number[]): void {}

let v0: ParamType<typeof f0>; // let v0: number
let v1: ParamType<typeof f1>; // let v1: string
let v2: ParamType<typeof f2>; // let v2: {}
let v3: ParamType<typeof f3>; // let v3: undefined
let v4: ParamType<typeof f4>; // let v4: number[]. Oops, ожидалось тип
number вместо number[]

// определяем новый тип, что бы разрешить последний случай

type WithoutArrayParamType<T> =
    T extends (p: (infer U)[]) => void ? U :
    T extends (p: infer U) => void ? U :
    undefined;


let v5: WithoutArrayParamType<typeof f4>; // let v5: number. Ok
```

Принципы определения переменных в условных типах продемонстрированные на примере функционального типа, идентичны и для объектных типов.

ts

```
type ParamType<T> = T extends { a: infer A, b: infer B } ? A | B :
undefined;

let v: ParamType<{ a: number, b:string }>; // let v: string | number
```

Глава 46

Readonly, Partial, Required, Pick, Record

Чтобы сделать повседневные будни разработчика немного легче, *TypeScript*, реализовал несколько предопределенных сопоставимых типов, как - `Readonly<T>`, `Partial<T>`, `Required<T>`, `Pick<T, K>` и `Record<K, T>`. За исключением `Record<K, T>`, все они являются так называемым *гомоморфными типами* (homomorphic types). Простыми словами, *гомоморфизм* — это возможность изменять функционал сохраняя первоначальные свойства всех операций. Если на данный момент это кажется сложным, то текущая глава покажет, что за данным термином не скрывается ничего сложного. Кроме того, в ней будет подробно рассмотрен каждый из перечисленных типов.

[46.0] Readonly<T> (сделать члены объекта только для чтения)

Сопоставимый тип `Readonly<T>` добавляет каждому члену объекта модификатор `readonly`, делая их тем самым только для чтения.

ts

```ts
// lib.es6.d.ts

type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};
```

Наиболее частое применение данного типа можно встретить при определении функций и методов параметры которых принадлежать к объектным типам. Поскольку объектные типы передаются по ссылке, то с высокой долей вероятности, случайное изменение члена объекта может привести к непредсказуемым последствиям.

ts

```ts
interface IPerson {
  name: string;
  age: number;
}

/**
 * Функция, параметр которой не
 * защищен от случайного изменения.
 *
 * Поскольку объектные типы передаются
 * по ссылке, то с высокой долей вероятности,
 * случайное изменение поля name нарушит ожидаемый
 * ход выполнения программы.
 */
function mutableAction(person: IPerson) {
  person.name = "NewName"; // Ok
}


/**
 * Надежная функция защищающая свои
 * параметры от изменения не требуя описания
 * нового неизменяемого типа.
 */
function immutableAction(person: Readonly<IPerson>) {
  person.name = "NewName"; // Error -> Cannot assign to 'name' because
it is a read-only property.
}
```

Тип сопоставления `Readonly<T>` является гомоморфным и добавляя свой модификатор `readonly` не влияет на уже существующие модификаторы. Сохранения исходным типом своих первоначальных характеристик (в данном случае — модификаторы), делает сопоставленный тип `Readonly<T>` гомоморфным.

ts

```
interface IPerson {
  gender?: string;
}

type Person = Readonly<IPerson> // type Person = { readonly gender?:
string; }
```

В качестве примера можно привести часто встречающейся на практике случай, в котором универсальный интерфейс описывает объект предназначенный для работы с данными. Поскольку в львиной доле данные представляются объектными типами, интерфейс декларирует их как неизменяемые, что в дальнейшем, при его реализации, избавит разработчика в типизации конструкций и тем самым сэкономив для него время на более увлекательные задачи.

ts

```
/**
 * Интерфейс необходим для описания экземпляра
 * провайдеров с которыми будет сопряженно
 * приложение. Кроме того, интерфейс описывает
 * поставляемые данные как только для чтения,
 *, что в будущем может сэкономить время.
 */
interface IDataProvider<OutputData, InputData = null> {
  getData(): Readonly<OutputData>;
}

/**
 * Абстрактный класс описание определяющий
 * поле data доступный только потомка как
 * только для чтения. Это позволит предотвратить
 * случайное изменение данных в классах потомках.
 */
abstract class DataProvider<InputData, OutputData = null> implements
IDataProvider<InputData, OutputData> {
  constructor(protected data?: Readonly<OutputData>) {
  }

  abstract getData(): Readonly<InputData>
}



interface IPerson {
  firstName: string;
  lastName: string;
}

interface IPersonDataProvider {
  name: string;
}
```

```
class PersonDataProvider extends DataProvider<IPerson,
IPersonDataProvider> {
    getData() {
      /**
       * Работая в теле потомков DataProvider
       * будет не так просто случайно изменить
       * данные доступные через ссылку this.data
       */
      let [firstName, lastName] = this.data.name.split(` `);
      let result = { firstName, lastName };

      return result;
    }

}


let provider = new PersonDataProvider({ name: `Ivan Ivanov` });
```

[46.1] Partial<T> (сделать все члены объекта необязательными)

Сопоставимый тип `Partial<T>` добавляет членам объекта модификатор `?:` делая их таким образом необязательными.

ts

```
// lib.es6.d.ts

type Partial<T> = {
    [P in keyof T]?: T[P];
};
```

Тип сопоставления `Partial<T>` является гомоморфным и не влияет на существующие модификаторы, а лишь расширяет модификаторы конкретного типа.

ts

```
interface IPerson {
  readonly name: string; // поле помечено, как только для чтения
}
```

```
/**
 * добавлен необязательны модификатор
 * и при этом сохранен модификатор readonly
 *
 * type Person = {
 *  readonly name?: string;
 * }
 */
type Person = Partial<IPerson>
```

Представьте приложение зависящее от конфигурации, которая как полностью, так и частично, может быть переопределена пользователем. Поскольку работоспособность приложения завязана на конфигурации, члены определенные в типе представляющем её, должны быть обязательными. Но поскольку пользователь может переопределить лишь часть конфигурации, функция выполняющая её слияние с конфигурацией по умолчанию, не может указать в аннотации типа уже определенный тип, так как его члены обязательны. Описывать новый тип, слишком утомительно. В таких случаях необходимо прибегать к помощи `Partial<T>` .

ts

```
interface IConfig {
  domain: string;
  port: "80" | "90";
}

const DEFAULT_CONFIG: IConfig = {
  domain: `https://domain.com`,
  port: "80"
};


function createConfig(config: IConfig): IConfig {
  return Object.assign({}, DEFAULT_CONFIG, config);
}

/**
 * Error -> Поскольку в типе IConfig все
 * поля обязательные, данную функцию
 * не получится вызвать с частичной конфигурацией.
 */
createConfig({
  port: "80"
});


function createConfig(config:Partial<IConfig>): IConfig {
  return Object.assign({}, DEFAULT_CONFIG, config);
}

/**
 * Ok -> Тип Partial<T> сделал все члены
```

```
 * описанные в IConfig необязательными,
 * поэтому пользователь может переопределит
 * конфигурацию частично.
 */
createConfig({
  port: "80"
});
```

[46.2] Required<T> (сделать все необязательные члены обязательными)

Сопоставимый тип `Required<T>` удаляет все необязательные модификаторы `?:` приводя члены объекта к обязательным. Достигается это путем удаления необязательных модификаторов при помощи механизма *префиксов - и +* рассматриваемого в главе “Оператор keyof, Lookup Types, Mapped Types, Mapped Types - префиксы + и -”.

ts

```
type Required<T> = {
    [P in keyof T]-?: T[P];
};
```

Тип сопоставления `Required<T>` является полной противоположностью типу сопоставления `Partial<T>` .

ts

```
interface IConfig {
  domain: string;
  port: "80" | "90";
}

/**
 * Partial добавил членам IConfig
 * необязательный модификатор ->
 *
 * type T0 = {
 *  domain?: string;
 *  port?: "80" | "90";
 * }
 */
type T0 = Partial<IConfig>;
```

```ts
/**
 * Required удалил необязательные модификаторы
 * у типа T0 ->
 *
 * type T1 = {
 *  domain: string;
 *  port: "80" | "90";
 * }
 */
type T1 = Required<T0>;
```

Тип сопоставления `Required<T>` является гомоморфным и не влияет на модификаторы отличные от необязательных.

ts

```ts
interface IT {
  readonly a?: number;
  readonly b?: string;
}

/**
 * Модификаторы readonly остались
 * на месте ->
 *
 * type T0 = {
 *  readonly a: number;
 *  readonly b: string;
 * }
 */
type T0 = Required<IT>;
```

[46.3] Pick (отфильтровать объектный тип)

Сопоставимый тип `Pick<T, K>` предназначен для фильтрации объектного типа ожидаемого в качестве первого параметра типа. Фильтрация происходит на основе ключей представленных множеством литеральных строковых типов ожидаемых в качестве второго параметра типа.

ts

```ts
// lib.es6.d.ts
```

```
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
};
```

Простыми словами, результатом преобразования `Pick<T, K>` будет являться тип состоящий из членов первого параметра идентификаторы которых указанны во втором параметре.

ts

```ts
interface IT {
  a: number;
  b: string;
  c: boolean;
}

/**
 * Поле "c" отфильтровано ->
 *
 * type T0 = { a: number; b: string; }
 */
type T0 = Pick<IT, "a" | "b">;
```

Стоит заметить, что в случае указания несуществующих ключей возникнет ошибка.

ts

```ts
interface IT {
  a: number;
  b: string;
  c: boolean;
}

/**
 * Error ->
 *
 * Type '"a" | "U"' does not satisfy the constraint '"a" | "b" | "c"'.
 * Type '"U"' is not assignable to type '"a" | "b" | "c"'.
 */
type T1 = Pick<IT, "a" | "U">;
```

Тип сопоставления `Pick<T, K>` является гомоморфным и не влияет на существующие модификаторы, а лишь расширяет модификаторы конкретного типа.

ts

```ts
interface IT {
  readonly a?: number;
  readonly b?: string;
  readonly c?: boolean;
}
```

```ts
/**
 * Модификаторы readonly и ? сохранены ->
 *
 * type T2 = { readonly a?: number; }
 */
type T2 = Pick<IT, "a">;
```

Пример, который самым первым приходит в голову, является функция `pick`, в задачу которой входит создавать новый объект путем фильтрации членов существующего.

ts

```ts
function pick<T, K extends string & keyof T>(object: T, ...keys: K[]) {
    return Object
        .entries(object) // преобразуем объект в массив [идентификатор,
значение]
        .filter(([key]: Array<K>) => keys.includes(key)) // фильтруем
        .reduce((result, [key, value]) => ({...result, [key]: value}),
{} as Pick<T, K>); // собираем объект из прошедших фильтрацию членов
}


let person = pick({
    a: 0,
    b: ``,
    c: true
}, `a`, `b`);

person.a; // Ok
person.b; // Ok
person.c; // Error -> Property 'c' does not exist on type 'Pick<{ a:
number; b: string; c: boolean; }, "a" | "b">'.
```

[46.4] Record<K, T> (динамически определить поле в объектном типе)

Сопоставимый тип `Record<K, T>` предназначен для динамического определения полей в объектном типе. Данный тип определяет два параметра типа. В качестве первого параметра ожидается множество ключей представленных множеством `string` или `Literal String` - `Record<"a", T>` или `Record<"a" | "b", T>`. В

качестве второго параметра ожидается конкретный тип данных, который будет ассоциирован с каждым ключом.

ts

```ts
// lib.es6.d.ts

type Record<K extends string, T> = {
    [P in K]: T;
};
```

Самый простой пример, который первым приходит в голову, это замена индексных сигнатур.

ts

```ts
/**
 * Поле payload определенно как объект
 * с индексной сигнатурой, что позволит
 * динамически записывать в него поля.
 */
interface IConfigurationIndexSignature {
  payload: {
    [key: string]: string
  }
}

/**
 * Поле payload определенно как
 * Record<string, string>, что аналогично
 * предыдущему варианту, но выглядит более
 * декларативно.
 */
interface IConfigurationWithRecord {
  payload: Record<string, string>
}


let configA: IConfigurationIndexSignature = {
  payload: {
    a: `a`,
    b: `b`
  }
}; // Ok
let configB: IConfigurationWithRecord = {
  payload: {
    a: `a`,
    b: `b`
  }
}; // Ok
```

Но в отличии от индексной сигнатуры типа `Record<K, T>` может ограничить диапазон ключей.

ts

```ts
type WwwConfig = Record<"port" | "domain", string>

let wwwConfig: WwwConfig = {
  port: "80",
  domain: "https://domain.com",

  user: "User" // Error -> Object literal may only specify known
properties, and 'user' does not exist in type 'Record<"port" |
"domain", string>'.
};
```

В данном случае было бы даже более корректным использовать `Record<K, T>` в совокупности с ранее рассмотренным типом `Partial<T>`.

ts

```ts
type WwwConfig = Partial<Record<"port" | "domain", string>>

let wwwConfig: WwwConfig = {
  port: "80",
                // Ok -> поле domain теперь не обязательное
  user: "User" // Error -> Object literal may only specify known
properties, and 'user' does not exist in type 'Record<"port" |
"domain", string>'.
};
```

Также не будет лишним упомянуть, что поведение данного типа при определении в объекте с предопределенными членами, идентификаторы которых ассоциированы с типами отличными от типа указанного в качестве второго параметра, идентично поведению индексной сигнатуры. Напомню, что при попытке определить в объекте члены идентификаторы которых будут ассоциированы с типами отличными от указанных в индексной сигнатуре, возникнет ошибка.

ts

```ts
/**
 * Ok -> поле a ассоциировано с таким
 * же типом, что указан в индексной сигнатуре.
 */
interface T0 {
  a: number;

  [key: string]: number;
}

/**
```

```
 * Error -> тип поля a не совпадает с типом
 * указанным в индексной сигнатуре.
 */
interface T1 {
  a: string; // Error -> Property 'a' of type 'string' is not
assignable to string index type 'number'.

  [key: string]: number;
}
```

Данный пример можно переписать с использованием типа пересечения.

ts

```
interface IValue {
  a: number;
}

interface IDynamic {
  [key: string]: string;
}


type T = IDynamic & IValue;

/**
 * Error ->
 * Type '{ a: number; }' is not assignable to type 'IDynamic'.
 * Property 'a' is incompatible with index signature.
 * Type 'number' is not assignable to type 'string'.
 */
let t: T = {
  a: 0,
}
```

Аналогичное поведение будет и для пересечения определяемого типом `Record<K, T>`.

ts

```
interface IValue {
  a: number;
}


type T = Record<string, string> & IValue;

/**
 * Error ->
 * Type '{ a: number; }' is not assignable to type 'Record<string,
string>'.
 * Property 'a' is incompatible with index signature.
 * Type 'number' is not assignable to type 'string'.
```

```
 */
let t: T = {
  a: 0,
}
```

Глава 47

Exclude, Extract, NonNullable, ReturnType, InstanceType, Omit

Чтобы сэкономить время разработчиков, в систему типов *TypeScript* были включены несколько часто требующихся условных типов, каждый из которых будут подробно рассмотрен в этой главе.

[47.0] Exclude<T, U> (исключает из T признаки присущие U)

В результате разрешения условный тип `Exclude<T, U>` будет представлять разницу типа `T` относительно типа `U`. Параметры типа `T` и `U` могут быть представлены как единичным типом, так и множеством `union`.

ts

```ts
// @filename: lib.d.ts

type Exclude<T, U> = T extends U ? never : T;
```

Простыми словами из типа `T` будут исключены признаки (ключи) присущие также и типу `U`.

ts

```
let v0: Exclude<number|string, number|boolean>; // let v0: string
let v1: Exclude<number|string, boolean|object>; // let v1: string|number
let v2: Exclude<"a" | "b", "a" | "c">; // let v2: "b"
```

В случае, если оба аргумента типа принадлежат к одному и тому же типу данных, `Exclude<T, U>` будет представлен типом `never`.

ts

```
let v4: Exclude<number|string, number|string>; // let v4: never
```

Его реальную пользу лучше всего продемонстрировать на реализации функции, которая на входе получает два разных объекта, а на выходе возвращает новый объект, состоящий из членов присутствующих в первом объекте, но отсутствующих во втором. Аналог функции `difference` из широко известной библиотеки *lodash*.

ts

```
declare function difference<T, U>(a: T, b: U): Pick<T, Exclude<keyof T,
keyof U>>


interface IA { a: number; b: string; }
interface IB { a: number; c: boolean; }


let a: IA = { a: 5, b: '' };
let b: IB = { a: 10, c: true };


interface IDifference { b: string; }


let v0: IDifference = difference(a, b); // Ok
let v1: IA = difference(a, b); // Error -> Property 'a' is missing in
type 'Pick<IA, "b">' but required in type 'IA'.
let v2: IB = difference(a, b); // Error -> Type 'Pick ' is missing the
following properties from type 'IB': a, c
```

[47.1] Extract<T, U> (общие для двух типов признаки)

В результате разрешения условный тип `Extract<T, U>` будет представлять пересечение типа `T` относительно типа `U`. Оба параметра типа могут быть представлены как обычным типом, так `union`.

ts

```ts
// @filename: lib.d.ts

type Extract<T, U> = T extends U ? T : never;
```

Простыми словами, после разрешения `Extract<T, U>` будет принадлежать к типу определяемого признаками (ключами) присущих обоим типам. То есть, тип `Extract<T, U>` является противоположностью типа `Exclude<T, U>`.

ts

```ts
let v0 :Extract<number|string, number|string>; // let v0: string |
number
let v1 :Extract<number|string, number|boolean>; // let v1: number
let v2 :Extract<"a" | "b", "a" | "c">; // let v2: "a"
```

В случае, когда общие признаки отсутствуют, тип `Extract<T, U>` будет представлять тип `never`.

ts

```ts
let v3 :Extract<number|string, boolean|object>; // let v3: never
```

Условный тип `Extract<T, U>` стоит рассмотреть на примере реализации функции принимающей два объекта и возвращающей новый объект, состоящий из членов первого объекта, которые также присутствуют и во втором объекте.

ts

```ts
declare function intersection<T, U>(a: T, b: U): Pick<T, Extract<keyof
T, keyof U>>
```

```ts
interface IA { a: number; b: string; }
interface IB { a: number; c: boolean; }


let a: IA = { a: 5, b: '' };
let b: IB = { a: 10, c: true };


interface IIntersection { a: number; }


let v0: IIntersection = intersection(a, b); // Ok
let v1: IA = intersection(a, b); // Error -> Property 'b' is missing in
type 'Pick<IA, "a">' but required in type 'IA'.
let v2: IB = intersection(a, b); // Error -> Property 'c' is missing in
type 'Pick<IA, "a">' but required in type 'IB'.
```

[47.2] NonNullable<T> (удаляет типы null и undefined)

Условный тип `NonNullable<T>` служит для исключения из типа признаков типов `null` и `undefined`. Единственный параметр типа может принадлежать как к обычному типу, так и множеству определяемого тип `union`.

ts

```ts
// @filename: lib.d.ts

type NonNullable<T> = T extends null | undefined ? never : T;
```

Простыми словами, данный тип удаляет из аннотации типа такие типы, как `null` и `undefined`.

ts

```ts
let v0: NonNullable<string | number | null>; // let v0: string | number
let v1: NonNullable<string | undefined | null>; // let v1: string
let v2: NonNullable<string | number | undefined | null>; // let v2:
string | number
```

В случае, когда тип, выступающий в роли единственного аргумента типа, принадлежит только к типам `null` и\или `undefined`, `NonNullable<T>` представляет тип `never`.

ts

```
let v3: NonNullable<undefined | null>; // let v3: never
```

[47.3] ReturnType<T> (получить тип значения возвращаемого функцией)

Условный тип `ReturnType<T>` служит для установления возвращаемого из функции типа. В качестве параметра типа должен обязательно выступать *функциональный тип*.

ts

```
// @filename: lib.d.ts

type ReturnType<T extends (...args: any) => any> = T extends (...args:
any) => infer R ? R : any;
```

На практике очень часто требуется получить тип к которому принадлежит значение возвращаемое из функции. Единственное на, что стоит обратить внимание, что в случаях, когда тип возвращаемого из функции значения является параметром типа, у которого отсутствуют хоть какие-то признаки, то тип `ReturnType<T>` будет представлен пустым объектным типом `{}`.

ts

```
let v0: ReturnType<() => void>; // let v0: void
let v1: ReturnType<() => number | string>; // let v1: string|number
let v2: ReturnType<<T>() => T>; // let v2: {}
let v3: ReturnType<<T extends U, U extends string[]>() => T>; // let
v3: string[]
let v4: ReturnType<any>; // let v4: any
let v5: ReturnType<never>; // let v5: never
let v6: ReturnType<Function>; // Error
let v7: ReturnType<number>; // Error
```

[47.4] InstanceType<T> (получить через тип класса тип его экземпляра)

Условный тип `InstanceType<T>` предназначен для получения типа экземпляра на основе типа представляющего класс. Параметр типа `T` должен обязательно принадлежать к *типу класса*.

ts

```ts
// @filename: lib.d.ts

type InstanceType<T extends new (...args: any) => any> = T extends new
(...args: any) => infer R ? R : any;
```

В большинстве случаев идентификатор класса задействован в приложении в качестве типа его экземпляра.

ts

```ts
class Animal {
  move(): void {}
}

/**
 * Тип Animal представляет объект класса,
 *, то есть его экземпляр полученный при
 * помощи оператора new.
 */
function f(animal: Animal){
  type Param = typeof Animal;

  // здесь Param представляет экземпляр типа Animal
}
```

Но сложные приложения часто требуют динамического создания своих компонентов. В таких случаях фабричные функции работают не с экземплярами классов, а непосредственно с самими классами.

Стоит напомнить, что в *JavaScript* классы, это всего-лишь *синтаксический сахар* над старой, доброй *функцией конструктором*. И как известно объект функции конструктора представляет объект класса содержащего ссылку на прототип, который и представляет экземпляр. Другими словами, в *TypeScript* идентификатор класса указанный в аннотации

типа, представляет описание прототипа. Чтобы получить тип самого класса, необходимо выполнить над идентификатором класса *запрос типа*.

ts

```ts
class Animal {
  move(): void {}
}

type Instance = Animal;
type Class = typeof Animal;

type MoveFromInstance = Instance["move"]; // Ok ->() => void
type MoveFromClass = Class["move"]; // Error -> Property 'move' does
not exist on type 'typeof Animal'.
```

Таким образом, грамотно вычислить тип экземпляра в фабричной функции можно при помощи типа `InstanceType<T>`.

ts

```ts
class Animal {
  move(): void {}
}

function factory(Class: typeof Animal){
  type Instance = InstanceType<Class>;

  let instance: Instance = new Class(); // Ok -> let instance: Animal
}
```

Хотя можно прибегнуть и к менее декларативному способу к запросу типа свойства класса `prototype`.

ts

```ts
function factory(Class: typeof Animal){
  type Instance = Class["prototype"];

  let instance: Instance = new Class(); // Ok -> let instance: Animal
}
```

И последнее о чем стоит упомянуть, что результат получение типа непосредственно через `any` и `never` будет представлен ими же. Остальные случаи приведут к возникновению ошибки.

ts

```ts
class Animal {}
```

```
let v0: InstanceType<any>; // let v0: any
let v1: InstanceType<never>; // let v1: never
let v2: InstanceType<number>; // Error
```

[47.5] Parameters<T> (получить тип размеченного кортежа описывающий параметры функционального типа)

Расширенный тип `Parameters<T>` предназначен для получения типов указанных в аннотации параметров функции. В качестве аргумента типа ожидается *функциональный тип*, на основе которого будет получен размеченный кортеж описывающий параметры этого функционального типа.

ts

```
type Parameters<T extends (...args: any[]) => any> = T extends (...args:
infer P) => any ? P : never;
```

`Parameters<T>` возвращает типы параметров в виде кортежа.

ts

```
function f<T>(p0: T, p1: number, p2: string, p3?: boolean, p4: object =
{}) {

}

/**
 * type FunctionParams = [p0: unknown, p1: number, p2: string, p3?:
boolean, p4?: object]
 */
type FunctionParams = Parameters<typeof f>;
```

[47.6] ConstructorParameters<T> (получить через тип класса размеченный кортеж описывающий параметры его конструктора)

Расширенный тип `ConstructorParameters<T>` предназначен для получения типов указанных в аннотации параметров конструктора.

ts

```ts
type ConstructorParameters<T extends new (...args: any[]) => any> = T
extends new (...args: infer P) => any ? P : never;
```

В качестве единственного параметра типа `ConstructorParameters<T>` ожидает тип самого класса, на основе конструктора которого будет получен размеченный кортеж описывающий параметры этого конструктора.

ts

```ts
class Class<T> {
    constructor(p0: T, p1: number, p2: string, p3?: boolean, p4: object
= {}) {}
}

/**
 * type ClassParams = [p0: unknown, p1: number, p2: string, p3?:
boolean, p4?: object]
 */
type ClassParams = ConstructorParameters<typeof Class>;
```

[47.7] Omit<T, K> (исключить из T признаки ассоциированными с ключами перечисленных множеством K)

Расширенный тип `Omit<T, K>` предназначен для определения нового типа путем исключения заданных признаков из существующего тип.

ts

```ts
// lib.d.ts

type Omit<T, K extends string | number | symbol> = {
    [P in Exclude<keyof T, K>]: T[P];
}
```

В качестве первого аргумента типа тип `Omit<T, K>` ожидает тип данных, из которого будут исключены признаки, связанные с ключами, переданными в качестве второго аргумента типа.

Простыми словами, к помощи `Omit<T, K>` следует прибегать в случаях необходимости определения типа, представляющего некоторую часть уже существующего типа.

ts

```
type Person = {
  firstName: string;
  lastName: string;

  age: number;
};

/**
 * Тип PersonName представляет только часть типа Person
 *
 * type PersonName = {
 *  firstName: string;
 *  lastName: string;
 * }
 */
type PersonName = Omit<Person, 'age'>; // исключение признаков
связанных с полем age из типа Person
```

Глава 48

Массивоподобные readonly типы, ReadonlyArray, ReadonlyMap, ReadonlySet

Чем меньше шансов случайного изменения значений определенных в объектных типах, тем больше программа защищена от ошибок во время выполнения. Очередным шагом в этом направлении стали неизменяемые массивоподобные типы `ReadonlyArray<T>`, `ReadonlyMap<K, V>`, `ReadonlySet<T>`, а также механизм указания модификатора `readonly` в аннотации типа.

[48.0] Массивоподобные readonly типы (модифицировать непосредственно в аннотации типа)

TypeScript реализует механизм позволяющий определять массивы и кортежи, как неизменяемые структуры данных. Для этого к типу указанному в аннотации типа добавляется модификатор `readonly`.

ts

```
let array: readonly string[] = ['Kent', 'Clark']; // Массив
let tuple: readonly [string, string] = ['Kent', 'Clark']; // Кортеж
```

Элементы массивоподобных структур, определенных как `readonly`, невозможно заменить или удалить. Кроме того, в подобные структуры невозможно добавить новые элементы. Иными словами, у массивоподобных `readonly` типов отсутствуют признаки предназначенные для изменения их содержимого.

В случае объявления `readonly` массива становится невозможно изменить его элементы с помощью индексной сигнатуры (`array[...]`)

ts

```
let array: readonly string[] = ['Kent', 'Clark']; // Массив
array[0] = 'Wayne'; // Error, -> Index signature in type 'readonly
string[]' only permits reading.
array[array.length] = 'Batman'; // Error ->  Index signature in type
'readonly string[]' only permits reading.
```

Помимо этого, у `readonly` массива отсутствуют методы, с помощью которых можно изменить элементы массива.

ts

```
let array: readonly string[] = ['Kent', 'Clark'];
array.push('Batman'); // Error ->  Property 'push' does not exist on
type 'readonly string[]'.
array.shift(); // Error -> Property 'shift' does not exist on type
'readonly string[]'.

array.indexOf('Kent'); // Ok
array.map(item => item); // Ok
```

С учетом погрешности на известные различия между массивом и кортежем, справедливо утверждать, что правила для `readonly` массива справедливы и для `readonly` кортежа. Помимо того, что невозможно изменить или удалить слоты кортежа, он также теряет признаки массива, которые способны привести к его изменению.

ts

```
let tuple: readonly [string, string] = ['Kent', 'Clark'];
tuple[0] = 'Wayne'; // Error -> Cannot assign to '0' because it is a
read-only property.

tuple.push('Batman'); // Error -> Property 'push' does not exist on
type 'readonly [string, string]'.
tuple.shift(); // Error -> Property 'shift' does not exist on type
'readonly [string, string]'.
```

```
tuple.indexOf('Kent'); // Ok
tuple.map(item => item); // Ok
```

Также не будет лишним упомянуть, что массив или кортеж указанный в аннотации с помощью расширенного типа `Readonly<T>`, расценивается выводом типов как помеченный модификатором `readonly`.

ts

```ts
// type A = readonly number[];
type A = Readonly<number[]>;

// type B = readonly [string, boolean];
type B = Readonly<[string, boolean]>;
```

Благодаря данному механизму в сочетании с механизмом множественного распространения (`spread`), становится возможным типизировать сложные сценарии, одним из которых является реализация известной всем функции `concat` способной объединить не только массивы, но и кортежи.

ts

```ts
type A = readonly unknown[];

function concat<T extends A, U extends A>(a: T, b: U): [...T, ...U] {
    return [...a, ...b];
}

// let v0: number[]
let v0 = concat([0, 1], [2, 3]);

// let v1: [0, 1, 2, 3]
let v1 = concat([0, 1] as const, [2, 3] as const);

// let v2: [0, 1, ...number[]]
let v2 = concat([0, 1] as const, [2, 3]);

// let v3: number[]
let v3 = concat([0, 1], [2, 3] as const);
```

Напоследок стоит упомянуть, что вывод типов расценивает `readonly` массив как принадлежащий к интерфейсу `ReadonlyArray<T>`, речь о котором пойдет далее.

[48.1] ReadonlyArray<T> (неизменяемый массив)

Расширенный тип `ReadonlyArray<T>` предназначен для создания неизменяемых массивов. `ReadonlyArray<T>` запрещает изменять значения массива, используя индексную сигнатуру `array[n]` .

ts

```ts
let array: ReadonlyArray<number> = [0, 1, 2];

array[0] = 1; // Error -> Index signature in type 'readonly number[]'
only permits reading.
array[array.length] = 3; // Error -> Index signature in type 'readonly
number[]' only permits reading.
```

Кроме того, тип `ReadonlyArray<T>` не содержит методы, способные изменить, удалить или добавить элементы.

ts

```ts
let array: ReadonlyArray<number> = [0, 1, 2];

array.push(3); // Error -> Property 'push' does not exist on type
'readonly number[]'.
array.shift(); // Error -> Property 'shift' does not exist on type
'readonly number[]'.

array.indexOf(0); // Ok
```

[48.2] ReadonlyMap<K, V> (неизменяемая карта)

Расширенный тип `ReadonlyMap<K, V>` , в отличие от своего полноценного прототипа, не имеет методов, способных его изменить.

ts

```
let map: ReadonlyMap<string, number> = new Map([["key", 0]]);
```

[48.3] ReadonlySet<T> (неизменяемое множество)

Аналогично другим структурам данных предназначенных только для чтения, расширенный тип `ReadonlySet<T>` не имеет методов, способных его изменить.

ts

```
let set: ReadonlySet<number> = new Set([0, 1, 2]);
```

Глава 49

Синтаксические конструкции и операторы

Кроме типизации, *TypeScript* пытается сделать жизнь разработчиков более комфортной за счет добавления синтаксического сахара в виде операторов не существующих в *JavaScript* мире. Помимо этого, текущая глава поведает о неоднозначных моментах связанных с уже хорошо известными, по *JavaScript*, операторами.

[49.0] Операторы присваивания короткого замыкания (&&=, ||=, &&=)

В большинстве языков, в том числе и *JavaScript*, существует такое понятие как составные операторы присваивания (*compound assignment operators*) позволяющие совмещать операцию присваивания при помощи оператора `=`, с какой-либо другой допустимой операции (`+-*/!` и т.д.) и тем самым значительно сокращать выражения.

ts

```ts
let a = 1;
let b = 2;

a += b; // тоже самое, что a = a + b
a *= b; // тоже самое, что a = a * b
// и т.д.
```

Множество существующих операторов совместимы с оператором `=` за исключением трех, таких часто применяемых операторов, как *логическое И* (`&&`), *логическое ИЛИ* (`||`) и оператор *нулевого слияния* (`??`).

ts

```ts
a = a && b;
a = a || b;
a = a ?? b;
```

Поскольку дополнительные синтаксические возможности лишь упрощают процесс разработки программ, благодаря комьюнити, в *TypeScript* появился механизм обозначаемый как *операторы присваивания короткого замыкания*. Данный механизм позволяет совмещать упомянутые ранее операторы `&&` , `||` и `??` непосредственно с оператором присваивания.

ts

```ts
let a = {};
let b = {};

a &&= b; // a && (a = b)
a ||= b; // a || (a = b);
a ??= b; // a !== null && a !== void 0 ? a : (a = b);
```

[49.1] Операнды для delete должны быть необязательными

Представьте случай при котором в *JavaScript* коде вам необходимо удалить у объекта одно из трех определенных в нем полей.

js

```js
let o = {
    a: 0,
    b: '',
    c: true
};

const f = o => delete o.b;

f(0); // удаляем поле b
```

```
Object
    .entries(o)
    .forEach( ([key, value]) => console.log(key, value) );
/**
 * log -
 * -> a, 0
 * -> b, true
 */
```

Задача предельно простая только с точки зрения динамической типизации *JavaScript*. С точки зрения статической типизации *TypeScript*, удаление члена объекта нарушает контракт представляемый декларацией типа. Простыми словами, *TypeScript* не может гарантировать типобезопасность, пока не может гарантировать существование членов объекта описанных в его типе.

ts

```
type O = {
    a: number;
    b: string;
    c: boolean;
}

let o: O = {
    a: 0,
    b: '',
    c: true
};

const f = (o: O) => delete o.b; // [*]

f(o); // удаляем поле b

/**
 * [*] Error ->
 * Объект o больше не отвечает
 * типу O поскольку в нем нет
 * обязательного поля b. Поэтому
 * если дальше по ходу выполнения
 * программы будут производится
 * операции над удаленным полем,
 *, то возникнет ошибка времени выполнения.
 */
```

Поэтому *TypeScript* позволяет удалять члены объекта при помощи оператора `delete` только в том случае, если они имеют тип `any`, `unknown`, `never` или объявлены как необязательные.

ts

```
type T0 = {
    field: any;
}

const f0 = (o: T0) => delete o.field; // Ok


type T1 = {
    field: unknown;
}

const f1 = (o: T1) => delete o.field; // Ok


type T2 = {
    field: never;
}

const f2 = (o: T2) => delete o.field; // Ok


type T3 = {
    field?: number;
}

const f3 = (o: T3) => delete o.field; // Ok


type T4 = {
    field: number;
}

const f4 = (o: T4) => delete o.field; // Error -> The operand of a
'delete' operator must be optional.
```

[49.2] Объявление переменных 'необязательными' при деструктуризации массивоподобных объектов

При активном рекомендуемым флаге `--noUnusedLocals`, компилятор выбрасывает ошибки, если переменные, объявленные при деструктуризации массивоподобных объектов, не задействованы в коде.

ts

```ts
function getPlayerControlAll(){
    return [()=>{}, ()=>{}];
}

/**
 * Где-то в другом файле
 */
function f(){
    /**
     * [*] Error -> 'stop' is declared but its value is never read.
     */
    let [stop /** [*] */, play] = getPlayerControlAll();

    return play;
}
```

Несмотря на то, что существует способ получать только необходимые значения, это не решит проблему семантики кода, поскольку идентификатор переменной является частью мозаики иллюстрирующей работу логики. И хотя в *TypeScript*, эту проблему можно решить и другими способами, они ни чем не смогут помочь скомпилированному в *JavaScript* коду.

ts

```ts
function getPlayerControlAll(){
    return [()=>{}, ()=>{}];
}

/**
 * Где-то в другом файле
 */
function f(){
```

```
    /**
     * Ошибки больше нет, поскольку первый, неиспользуемый
     * элемент пропущен. Но не смотря на это, семантически становится
     * не понятно, что же возражает функция getPlayerControlAll().
     *
     * И несмотря на способы способные решить проблему в TypeScript,
     * в скомпилированном виде, от них не останется и следа.
     */
    let [, play] = getPlayerControlAll();

    return play;
}
```

Для таких случаев, в *TypeScript* существует возможность, при деструктуризации массивоподобных объектов, объявлять переменные, как *необязательные*. Что бы переменная расценивалась компилятором, как *необязательная*, её идентификатор должен начинаться с нижнего подчёркивания `_identifier`.

ts

```
function getPlayerControlAll(){
    return [()=>{}, ()=>{}];
}

/**
 * Где-то в другом файле
 */
function f(){
    /**
     * [*] Ok -> несмотря на то, что переменная stop не
     * задействована в коде, ошибки не возникает, что позволяет
     * более глубоко понять логику кода.
     */
    let [_stop /** [*] */, play] = getPlayerControlAll();

    return play;
}
```

[49.3] Модификатор abstract для описания типа конструктора

Абстрактные классы предназначены исключительно для расширения (невозможно создать его экземпляр с помощью оператора `new`), а его абстрактные члены должны обязательно должны быть переопределены потомками.

ts

```
/**
 * Абстрактный класс с одним абстрактным методом.
 */
abstract class Shape {
    abstract getRectangle(): ClientRect;
}

/**
 * Из-за того, что класс абстрактный не получится создать его экземпляр
с помощью оператора new.
 */
new Shape(); // Error -> Cannot create an instance of an abstract
class.ts(2511)


/**
 * [0] Кроме этого, подкласс обязательно должен переопределять
абстрактные члены своего суперкласса.
 */
class Circle extends Shape {
    getRectangle(){// [0]
        return {
            width:0,
            height: 0,
            top: 0,
            right: 0,
            bottom: 0,
            left: 0
        };
    }
}
```

Но правила, с помощью которых компилятор работает с абстрактными классами, делают типы абстрактных и конкретных конструкторов несовместимыми. Другими словами, абстрактный класс нельзя передать по ссылке ограниченной более общим типом.

ts

```ts
interface IHasRectangle {
    getRectangle(): ClientRect;
}

type HasRectangleClass = new() => IHasRectangle;

/**
 * [*] Type 'typeof Shape' is not assignable to type
'HasRectangleClass'.
 Cannot assign an abstract constructor type to a non-abstract
constructor type.ts(2322)
 */
let ClassType: HasRectangleClass = Shape; // Error [*]
```

Кроме этого, невозможно получить тип экземпляра абстрактного класса с помощью вспомогательного типа `InstanceType<T>` .

ts

```ts
/**
 * [*] Type 'typeof Shape' does not satisfy the constraint 'new
(...args: any) => any'.
  Cannot assign an abstract constructor type to a non-abstract
constructor type.ts(2344)
 */
type Instance = InstanceType<typeof Shape>; // Error [*]
```

Это в свою очередь не позволяет реализовать механизм динамического наследования.

ts

```ts
function subclassCreator(Base: new() => IHasRectangle){
    return class extends Base {
        getRectangle(){
            return {
                width:0,
                height: 0,
                top: 0,
                right: 0,
                bottom: 0,
                left: 0
            };
        }
    }
```

```
}

/**
 * [*] Argument of type 'typeof Shape' is not assignable to parameter
of type 'new () => IHasRectangle'.
 Cannot assign an abstract constructor type to a non-abstract
constructor type.ts(2345)
 */
subclassCreator(Shape); // Error [*] -> передача в качестве аргумента
абстрактный класс
subclassCreator(Circle); // Ok -> передача в качестве аргумента
конкретный класс
```

Для решения этой проблемы, в *TypeScript* существует модификатор `abstract` предназначенный для указания в описании типа конструктора.

ts

```
interface IHasRectangle {
    getRectangle(): ClientRect;
}

type HasRectangleClass = abstract new() => IHasRectangle;


let ClassType: HasRectangleClass = Shape; // Ok
```

Несмотря на то, что тип класса имеет абстрактный модификатор, он также остается совместимым с типами конкретных классов.

ts

```
function subclassCreator(Base: abstract new() => IHasRectangle){
    return class extends Base {
        getRectangle(){
            return {
                width:0,
                height: 0,
                top: 0,
                right: 0,
                bottom: 0,
                left: 0
            };
        }
    }
}

subclassCreator(Shape); // Ok -> абстрактный класс
subclassCreator(Circle); // Ok -> конкретный класс
```

Также с помощью данного оператора можно реализовать собственный вспомогательный тип, позволяющий получить тип экземпляра.

ts

```ts
type AbstractInstanceType<T extends abstract new (...args: any) => any>
= T extends new (...args: any) => infer R ? R : any;

type Instance = AbstractInstanceType<typeof Shape>; // Ok
```

Глава 50

Типизированный React

С данной главы начинается серия посвященная детальному рассмотрению темы связанной с популярной библиотекой рендера - *React* и поскольку она предусмотренна как водная, то расскажет лишь о создании условий необходимых для компиляции `react` приложения.

[50.0] Расширение .tsx

React — это библиотека для создания пользовательских интерфейсов от компании *Facebook*. В основе библиотеки продвинутого рендера *React* лежит компонентный подход, для улучшения которого, стандартный синтаксис *JavaScript* был расширен *XML*-подобным синтаксисом. Таким образом свет увидело новое расширение `.jsx` .

Из-за высокой популярности *React* данное расширение получило свое типизированное представление в виде `.tsx` . Но для того, что бы компилятор *tsc* смог компилировать `.tsx` синтаксис, необходимо установить его опцию `--jsx` в одно из значений - `“react”` для вэб и настольных, а также `“react-native”` для мобильных приложений. По умолчанию выставлено значение `“preserve”` .

json

```
// @filename: tsconfig.json

{
    "compilerOptions": {
        "jsx": "react"
```

```
    }
}
```

Кроме того, помимо самого `react`, необходимо установить его декларации.

sh

```
npm i -D @types/react @types/react-dom
```

Поскольку на данный момент известно, что *TypeScript*, это всего-лишь надстройка над *JavaScript*, то единственный вопрос, способный возникнуть при первом знакомстве с типизированным *React* - "как правильно аннотировать ту или иную конструкцию". Поэтому цель общей темы, посвященной типизированному *React*, будет заключаться в подробном рассмотрении всех случаев способных возникнуть на начальном этапе.

Также необходимо уточнить, что возможности *TypeScript* позволяют аннотировать языковые конструкции как в классическом, так и в минималистическом типизированном стиле. В первом случае, аннотацию типа содержит каждая конструкция. Во втором, максимально допустимая часть работы перекладывается на *вывод типов*. Так как предполагается, что читатели хотят получить полную картину и, кроме того, возвращаться к произвольным частям материала в будущем, то весь код будет написан в классическом стиле. Кроме того, стоит оговорить, что на момент написание этой главы используется *React* `v16.13.1` и *TypeScript* `v4.1`. Поэтому, если вы заметили несоответствия, скорее поспешите уведомить об этом любым из возможных способов.

Глава 51

Функциональные компоненты

Поскольку создавать *React* приложения рекомендуется на основе функциональных компонентов, то именно с них и начнется наше погружение в типизированные `.tsx` конструкции.

Всем известно, что *React* компоненты, обозначаемые как *функциональные*, являются обычными функциями. И как все функции в *JavaScript*, они также могут быть определены двумя способами — в виде обычной функции (*Function Declaration*) и в виде функционального выражения (*Function Expression*), таящего один неочевидный нюанс, который подробно будет рассмотрен по ходу знакомства с ним.

[51.0] Определение компонента как Function Declaration

Факт, что функциональный компонент, является обычной функцией, предполагает необходимость в типизировании её сигнатуры или точнее её параметров, так как указание типа возвращаемого значения не просто можно, а даже рекомендуется опустить, возложив эту работу на вывод типов.

ts

```ts
/**[0] */
import React from "react";

/**[1] */
```

```
function Timer(/**[2] */) /**[3] */ {
  return <div>Is Timer!</div>;
}

export default Timer;

/**
 * [1] функциональный компонент
 * определенный как Function Declaration.
 * [2] отсутствующие параметры.
 * [3] отсутствие явного указания типа
 * возвращаемого типа.
 * [0] все компоненты обязаны импортировать
 * пространство имен React.
 */
```

Также необходимо всегда помнить, что независимо от того используется пространство имен `React` непосредственно или нет, модули определяющие *React* компоненты обязаны его импортировать. В противном случае компилятор напомнит об этом с помощью ошибки.

ts

```
/**
 * Error ->
 * 'React' refers to a UMD global, but the current file is a module.
 * Consider adding an import instead.
 */

export function Message() {
  return <span>I ❤️ TypeScript!</span>;
}
```

ts

```
import React from "react";

/**
 * Ok -> добавлен импорт пространства имен React
 */

export function Message() {
  return <span>I ❤️ TypeScript!</span>;
}
```

При определении первого параметра функционального компонента `props` появляется потребность в типе описывающем их.

ts

```
/**[0] */
interface Props {
  message?: string; /**[1] */
  duration: number; /**[2] */
}

                                                /**[3] */
function Timer({duration, message = `Done!`}: Props) {
  return <div></div>;
}

/**
 * [0] определения типа интерфейса
 * который будет указан в аннотации
 * первого параметра функционального
 * компонента [3] и чье описание
 * включает необязательное поле message [1]
 *  и обязательного duration [2]
 */
```

Поскольку идеология *React* подразумевает *прокидывание пропсов* из одного компонента в другой, то компоненту выступающему в роли провайдера помимо своих пропсов, необходимо описывать пропсы своих детей. В случаях, когда в пробросе нуждаются только несколько значений принадлежащих к типам из системы типов *TypeScript*, декларирование их в пропсах провайдера не будет выглядеть удручающе.

ts

```
// file Informer.tsx

/**[0] */
interface Props {
    message: string;
}

/**
 * [0] описание пропсов компонента Informer
 */

export default function Informer({message}: Props){
  return <h1>{message}</h1>
}



// file InformerDecorator.tsx

import Informer from "./Informer";


/**[0] */
```

```
interface Props {
    decor: number; /**[1] */
    message: string; /**[2] */
}

/**
 * [0] описание пропсов компонента InformerDecorator
 * [1] значение предназначаемое непосредственно текущему компоненту
 * [2] значение предназначаемое компоненту Informer
 */

export default function InformerDecorator({decor, message}: Props){
  return <Informer message={message + decor}/>;
}
```

Но если пробрасываемые пропсы представлены множеством значений, к тому же, частично или полностью принадлежат к пользовательским типам, то более целесообразно включить в описание типа пропсов компонента-провайдера, тип, описывающий пропсы компонента, которому они предназначаются. Поэтому тип описывающий пропсы желательно всегда экспортировать. Для того, что бы избежать коллизий именования типов представляющих пропсы, их идентификаторы необходимо конкретизировать, то есть давать более уникальные имена. Поэтому принято имени `Props` добавлять префикс идентичный названию самого компонента.

ts

```
// file Informer.tsx

/**[0] */           /**[1] */
export interface InformerProps {
    message: string;
}

/**
 * [0] экспорт типа
 * [1] уточнение идентификатора (имени)
 */

export default function Informer({message}: InformerProps){
  return <h1>{message}</h1>
}



// file InformerDecorator.tsx
                    /**[0] */
import Informer, {InformerProps} from "./Informer";

/**
 * [0] импорт типа пропсов
 */
```

```
/**[1] */                /**[2] */                    /**[3] */
export interface InformerDecoratorProps extends InformerProps {
    decor: number;
    /**[4] */
}

/**
 * [1] экспорт типа
 * [2] уточнение идентификатора (имени)
 * [3] расширение типа пропсов другого компонента
 * позволяет не дописывать необходимые ему поля [4]
 */


export default function InformerDecorator({decor, message}:
InformerDecoratorProps){
  return <Informer message={message + decor}/>;
}
```

В случаях когда компонент-провайдер нуждается только в части пропсов определенных в типе представляющих их, ненужную часть можно исключить с помощью типа `Omit<T, K>` или `Exclude<T, U>` .

Тем, кому ближе минимализм, может прийтись по душе подход с получением типа пропсов без его экспорта.

ts

```
// @filename: Informer.tsx
import React from "react";

// InformerProps не экспортируется наружу
interface InformerProps {
    message: string;
}

export default function Informer({message}: InformerProps){
  return <h1>{message}</h1>
}
```

ts

```
// @filename: InformerDecorator.tsx

                  /**[0] */
import React, { ComponentType } from "react";
    /**[1] */
import type Informer from "./Informer";

    /**[2] */  /**[3] */        /**[4] */   /**[5] */   /**[6] *//**[7]
*/
type GetProps<T> = T extends ComponentType<infer Props> ? Props :
```

```
unknown;

        /**[8] */                /**[9] */
type InformerProps = GetProps<typeof Informer>;

                                                    /**[10] */
export interface InformerDecoratorProps extends InformerProps {
    decor: number;
}

/**
 * [0] Импортируем обобщенный тип ComponentType<Props>
 * представляющий объединение классового и функционального
 * компонента. [1] Импортируем как "только тип" функциональный
 * компонент Informer. [2] Определяем тип GetProps<T> параметр типа
 * которого ожидает тип React компонента. Далее, с помощью механизма
 * определения типа на основе условия (условный тип) выясняем
принадлежит
 * ли тип [3] T к React компоненту и в этот момент определяем переменную
 * infer Props [5], которая и будет представлять тип пропсов компонент
T.
 * Если условие верно, то возвращаем тип [6] Props, иначе unknown.
 * [8] С помощью типа GetProps, на основе типа Informer, полученного
 * с помощью запроса типа [9], определяем новый  тип InformerProps,
 * который в дальнейшем используем по назначению.
 */
```

Подобный способ будет не заменим при работе со сторонними библиотеками *React* компонентов, которые не имеют экспорты типов описывающих свои пропсы.

Не будет лишним напомнить, что при помощи модификатора `readonly` не удастся избежать изменений переменных ссылки на которые были получены с помощью механизма деструктуризации.

ts

```
export interface InformerProps {
    /**[0] */
    readonly message: string;
};

                                            /**[1] */
export default function Informer({message} : Readonly<InformerProps>){
    message = 'new value'; /**[2] */

    return <h1>{message}</h1>
}

/**
 * [0] добавление модификатора readonly вручную,
 * а затем ещё тоже самое с помощью тип Readonly<T> [1]
```

```
 * и тем не менее переменная message изменяема [2].
 */
```

Такое поведение является причиной того, что деструктурированные идентификаторы являются определением новой переменной, а переменные не могут иметь модификатор `readonly` .

ts

```ts
/**[0] */
let o: Readonly<{f: number}> = {f: 0};
o.f = 1; // Error -> [1]

let {f} = o;
f = 1; // Ok -> [2]

/**
 * Определение переменной o с типом инлайн интерфейса [0]
 * поля которого модифицированный с помощью типа Readonly<T>.
 * При попытке изменить член o.f с модификатором readonly
 * возникает ошибка [1] ->
 * Cannot assign to 'f' because it is a read-only property.
 * Чего не происходит при изменении переменной определенной
 * в процессе деструктуризации.
 *
 * Механизм деструктуризации предполагает создание новой
 * переменной со значением одноименного члена объекта указанного
 * в качестве правого операнда выражения.
 * Выражение let {f} = o; эквивалентно выражению let f = o.f;
 * В этом случае создается новая переменная тип которой устанавливается
 * выводом типов. А вот модификатор readonly не применим к переменным.
 */
```

При необходимости декларирования `children` можно выбрать несколько путей. Первый из них подразумевает использование обобщенного типа `PropsWithChildren <P>` ожидающего в качестве аргумента типа тип представляющий пропсы. Данный тип определяет `children` как необязательное поле принадлежащее к `ReactNode` . При отсутствии продуманного плана насчёт `children` или необходимости их принадлежности к любому допустимому типу, данный тип будет как нельзя к месту.

ts

```ts
/**[0] */
import React, {PropsWithChildren} from "react";

export interface LabelProps {

}

                                                /**[1] */
export default function Label({children}: PropsWithChildren<LabelProps>)
```

```
{
    return (
        <span>{children}</span>
    );
}

/**
 * [0] импорт типа PropsWithChildren<T>
 * для указания его в аннотации типа параметров [1].
 */


<Label>{"label"}</Label>; // string as children -> Ok [2]
<Label>{1000}</Label>; // number as children -> Ok [3]
<Label></Label>; // undefined as children -> Ok [4]

 /**
  * При создании экземпляров компонента Label
  * допустимо указывать в качестве children
  * как строку [2], так и числа [3] и кроме
  * того не указывать значения вовсе [4]
  */
```

Если логика компонента предполагает обязательную установку `children` или уточнение типа к которому они принадлежат, то появляется необходимость их непосредственной декларации в типе представляющем пропсы этого компонента.

ts

```
/**
 * [0] children определены как
 * обязательное принадлежащее к
 * типу string поле.
 */
interface LabelProps {
    children: string; /**[0] */
}


export default function Label({children}: LabelProps){
    return (
        <span>{children}</span>
    );
}


<Label>{`label`}</Label>; // Ok
<Label>{1000}</Label>; // Error -> number не совместим со string
<Label></Label>; // Error -> children обязательны
```

Может показаться, что конкретизация типа `children` будет полезна при создании собственных `ui` компонентов. К примеру при создании компонента `List` выполняющего отрисовку элемента `ul`, было бы здорово определить `children`, как массив компонентов `ListItem` отрисовывающих элемент `li`.

Для этого понадобится импортировать обобщенный тип `ReactElement<P, T>`, первый параметр типа которого ожидает тип пропсов, а второй строку или конструктор компонента для указания его в качестве типа поля `type` необходимого для идентификации. По факту тип `ReactElement<P, T>` представляет экземпляр любого компонента в системе типов *React*. После определения компонентов `List` и `ListItem`, для первого нам понадобится переопределить поле `children`, указав ему тип `ReactElement<ListItemProps>`, что буквально означает — экземпляр компонента, пропсы которого принадлежат к типу указанному в качестве первого аргумента типа.

ts

```ts
/**[0] */
import React, { ReactElement, ReactNode } from "react";

/**
 * [0] импорт типа представляющего
 * экземпляр любого компонента.
 */

interface ListItemProps {
    children: ReactNode; /**[1] */
}

/**
 * [1] для примера определим тип children
 * как ReactNode представляющего любой
 * допустимый тип.
 */

 function ListItem({children}: ListItemProps){
 return <li>{children}</li>;
 }



interface ListProps {
    children: ReactElement<ListItemProps>; /**[2] */
}

/**
 * [2] при определении children
 * указываем тип ReactElement<ListItemProps>
 *, что стоит понимать как - экземпляр компонента
 * пропсы которого совместимы с типом ListItemProps.
 */

function List({children}: ListProps) {
```

```
    return <ul>{children}</ul>;
}

/**[3] */
<List>
    <ListItem>first</ListItem>
</List>

/**
 * [3] создаем экземпляр List
 * и указываем ему в качестве children
 * один экземпляр ListItem.
 */
```

Если кажется просто, то не стоит сомневаться, оно так и есть. Совершенно ничего сложного. Единственное стоит уточнить два важных момента.

Первый момент заключается в том, что конкретизация типа `children` для *React* элементов не работает. Проще говоря, если определить новый компонент `Label` и указать ему в качестве пропсов тип определяющий единственное поле `type`, то его экземпляр без возникновения ошибки можно будет указать в качестве `children` компоненту `List`.

ts

```
/**[0] */
interface ListItemProps {
    children: string;
}

/**
 * [0] оставляем тип пропсов,
 * но для упрощения удаляем компонент
 * нуждающийся в нем.
 */

/**[1] */
interface LabelProps {
    type: 'danger' | `error`;
    children: ReactNode;
}

/**[1] */
function Label({type, children}: LabelProps){
    return <span className={type}>{children}</span>
}

/**
 * [1] определяем компонент Label
 * и описываем его пропсы.
 */
```

```
interface ListProps {
    children: ReactElement<ListItemProps>; /**[2] */
}

/**
 * Тип children по прежнему указан
 * как ReactElement<ListItemProps>.
 */

// компонент List удален для упрощения

/**[3] */
<List>
    <Label type={'danger'}>Hello World!</Label>
</List>

/**
 * [3] несмотря на то, что в компоненте List
 * тип children обозначен как ReactElement<ListItemProps>
 * вместо ожидаемого экземпляра без возникновения
 * ошибки устанавливается тип ReactElement<LabelProps>.
 */
```

Всё дело в том, что экземпляр компонента представляется типом `Element` из пространства имен *JSX*, является производным от типа `ReactElement<P, T>` . Кроме того, при расширении, своему базовому классу, в качестве аргументов типа, он устанавливает `any` - `Element extends ReactElement<any, any>` . Это в свою очередь означает?, что любые экземпляры компонентов будут совместимы с любыми типами `ReactElement<P, T>` , что делает уточнение типа бессмысленным.

ts

```
let listItem = <ListItem>first</ListItem>; // let listItem: JSX.Element
let label = <Label type={'danger'}>label</Label>; // let label:
JSX.Element

/**
 * Поскольку оба экземпляра принадлежат
 * к типу JSX.Element который в свою очередь
 * является производным от типа ReactElement<any, any>,
 *, то любой экземпляр будет совместим с любым типом ReactElement<P, T>.
 */

 let v0: ReactElement<ListItemProps> = label;
 let v1: ReactElement<LabelProps> = listItem;
```

Кроме этого, `ReactElement<P, T>` совместим не только с экземплярами компонентов, но и *React* элементов.

ts

```ts
let v: ReactElement<ListItemProps> = <span></span>; // Ok
```

Это, как уже было сказано, делает конкретизацию `children`, для экземпляров *React* компонентов и элементов, бессмысленной. Остается надеяться, что это исправят.

Второй неочевидный момент состоит в том, что при текущей постановке, в случае необходимости указать в качестве `children` множество экземпляров `ListItem`, возникнет ошибка.

ts

```ts
interface ListProps {
    children: ReactElement<ListItemProps>; /**[0] */
}

/**[1] */
<List>
    <ListItem>first</ListItem>
    <ListItem>second</ListItem>
</List>

/**
 * [0] тип указан как единственный
 * экземпляр компонента. поэтому при
 * установке множества экземпляров [1]
 * возникает ошибка.
 */
```

Для разрешения подобного случая необходимо указать тип `children` как объединение (`Union`) определяемое типами представляющих как единственный экземпляр `ReactElement<ListItemProps>` так и множество, а если быть конкретнее, то массив экземпляров `ReactElement<ListItemProps>[]`.

ts

```ts
interface ListProps {
    children: ReactElement<ListItemProps> |
ReactElement<ListItemProps>[]; /**[0] */
}

/**[1] */
<List>
    <ListItem>first</ListItem>
    <ListItem>second</ListItem>
</List>

/**
 * [0] указание в качестве типа children
 * объединение предполагающее как единственный
```

```
 * экземпляр так и множество, ошибка [1] не возникает.
 */
```

Ввиду того, что тема касающаяся первого параметра функционального компонента обозначаемого пропсами, себя исчерпала, пришло время рассмотреть его второй, нечасто используемый, параметр, обозначаемый как *реф* и предназначенный для передачи ссылки на текущий компонент. Для большего понимания этого механизма необходимо начать его рассмотрение из далека. Но прежде стоит уточнить два момента. Во-первых, тему посвященную рефам стоит начать с напоминания, что они делятся на два вида. Первый вид предназначен для получения ссылок на *React элементы*, а второй на *React компоненты*. Так вот второй параметр функционального компонента предназначен для установления рефы на себя самого, тио есть на компонент. Во вторых, при рассмотрении механизма посвященного рефам, применяются хуки, чья логика работы будет затронута лишь поверхностно, поскольку впереди ожидает целая глава посвященная их подробному рассмотрению.

Представьте сценарий при котором форма должна перейти в начальное состояние путем вызова нативного метода `reset()`, что потребует получения на нее ссылки с помощью объекта рефы создаваемого хуком `useRef()`.

Для начала необходимо с помощью универсальной функции `useRef()` создать объект рефы и присвоить ссылку на него переменной, которую в дальнейшем установить элементу формы. Сразу стоит обратить внимание, что декларации *React* элементов содержат устаревшие типы в аннотации поля `ref`. Это непременно приведет к возникновению ошибки при установлении объекта рефы. Чтобы этого избежать, необходимо явным образом, при определении объекта рефы, преобразовать его к обобщенному типу `RefObject<T>`, которому в качестве аргумента типа установить тип нативного элемента, в данном случае `HTMLFormElement`. Также стоит сделать акцент на том, что необходимо именно преобразование. Указания аннотации типа переменной или передачи типа в качестве аргумента типа, хуку не поможет. Более детально поведение хука `useState()` рассматривается в главе посвященной предопределенным хукам.

ts

```ts
import React, { useRef, RefObject } from "react";

function Form(){
                    /**[0]          [1]           [2] */
    let formRef = useRef() as RefObject<HTMLFormElement>;

                  /**[3] */
    return <form ref={formRef}></form>
}

/**
 * Создаваемый хуком объект рефы необходимо преобразовать в обобщенный
тип
 * RefObject<T> [1] в качестве аргумента которому требуется указать
нативный тип
```

```
 * формы [2]. Если не произвести преобразования, то в момент установки
объекта рефы форме [3]
 * возникнет ошибка, поскольку декларации описывающие React элементы
содержат устаревший
 * тип в аннотации поля ref.
 *
 * БОЛЕЕ ПОДРОБНО В ТЕМЕ ПОСВЯЩЕННОЙ ХУКУ useRef()
 *
 */
```

Если появится необходимость задать форме первоначальное состояние извне компонента, то можно прибегнуть к механизму получения ссылки на сам компонент, или точнее на определяемый им объект выступающий в роли публичного *api*. Но, что бы стало возможным получить ссылку на функциональный компонент, его необходимо преобразовать с помощью универсальной функции `forwardRef<R, P>()`, на которой сфокусируется дальнейшее повествование.

По факту логика работы универсальной функции `forwardRef<R, P>(render)` заключатся в проверке единственного параметра на принадлежность к функциональному типу у которой помимо первого параметра представляющего `props`, определен ещё и второй, представляющий `ref`. Данная функция обозначается как `render` и теоретически её можно считать функциональным компонентом с определением второго параметра предназначенного для установления рефы. В системе типов *React* функция `render` принадлежит к обобщенному функциональному типу `ForwardRefRenderFunction<T, P>` определяющего два параметра типа, первый из которых представляет тип рефы, а второй пропсов. Первый параметр функции `render` представляющий пропсы не таит в себе ничего необычного. В отличии от него, второй параметр представляющий рефу, требует детального рассмотрения, поскольку именно с ним связан один неочевидный момент.

Дело в том, что рефы могут быть представлены как экземпляром объекта принадлежащего к типу `RefObject<T>` или полностью совместимым с ним `MutableRefObject<T>`, так и функцией `<T>(instance: T) => void`. Учитывая этот факт, функция `render`, в аннотации типа второго параметра `ref`, просто вынуждена указать все эти типы в качестве `union`. Но сложность состоит в том, что определение объединения происходит непосредственно в аннотации типа параметра `ref`. Простыми словами система типов *React* не предусмотрела более удобного и короткого псевдонима типа представляющего рефу определяемую функциональным компонентом.

ts

```
interface ForwardRefRenderFunction<T, P = {}> {
                                        /**
[0]                                  */
    (props: PropsWithChildren<P>, ref: ((instance: T | null) => void) |
MutableRefObject<T | null> | null): ReactElement | null;
}

/**
 * [0] тип объединение определенный
 * непосредственно в аннотации типа.
```

```
 * Простыми словами он не имеет более
 * удобного короткого псевдонима.
 */
```

Это означает, что функциональный компонент определенный как *Function Declaration* и указавший принадлежность второго параметра к типу, скажем `MutableRefObject<T>` не сможет пройти проверку на совместимость типов в качестве аргумента универсальной функции `forwardRef()`, даже если установить её аргументы типа. И причина тому контравариантность параметров функции при проверке на совместимость.

ts

```ts
/**[0]           [1] */
import React, { MutableRefObject, forwardRef } from "react";

export interface FormProps {}

/**[2] */
export interface FormApi {
    reset: () => void;
}

/**
 * [2] объявление типа описывающего доступное
 * api компонента.
 */
                                /**[3]       [4]         [5] */
function Form(props: FormProps, ref: MutableRefObject<FormApi>) {
    return null;
}

        /**[6]          [7]      [8]       [9]     [10] */
const FormWithRef = forwardRef<FormApi, FormProps>(Form /**Error */);

export default FormWithRef; /**[11] */

/**
 * [0] импорт обобщенного типа MutableRefObject<T>
 * который будет указан в аннотации типа [4] второго
 * параметра [3] функционального компонента предварительно
 * получив в качестве аргумента типа тип нативного dom
 * элемента HTMLDivElement [5].
 *
 * Несмотря на все принятые меры по типизации сигнатуры функционального
 * компонента Form избежать возникновения ошибки [10] при проверке на
совместимость
 * в момент передачи в качестве аргумента универсальной функции
forwardRef [7] не получится
 * даже при конкретизации с помощью аргументов функционального типа [8]
[9].
 *
```

```
 * [11] для экспорта функционального компонента определяющего второй
параметр необходимо
 * сохранить результат выполнения функции forwardRef [6].
 */
```

Разрешить данную ситуацию можно несколькими способами. Первый заключается в явном преобразовании типа функционального компонента к типу `ForwardRefRenderFunction<T, P>` которому в качестве аргументов типа требуется указать необходимые типы. При этом отпадает нужда в указании аргументов типа непосредственно самой универсальной функции `forwardRef<T, P>()`;

ts

```
/**[0] */
import React, { MutableRefObject, forwardRef, ForwardRefRenderFunction }
from "react";

export interface FormProps {}
export interface FormApi {
    reset: () => void;
}

function Form(props: FormProps, ref: MutableRefObject<FormApi>) {
    return null;
}

                        /**[6]                              [7]
[8]        [9] */
const FormWithRef = forwardRef(Form as ForwardRefRenderFunction<FormApi,
FormProps>);

export default FormWithRef;

/**
 * [0] импорт обобщенного функционального типа
ForwardRefRenderFunction<T, P>
 * к которому тип Form [7] будет преобразован с помощью оператора as,
для
 * чего потребуется указать необходимые аргументы типа [8] [9]. При
этом отпадает
 * потребность в установке аргументов непосредственно универсальной
функции [6]
 */
```

Следующий способ заключается в получении типа представляющего рефу непосредственно из самого функционального типа `ForwardRefRenderFunction<T, P>`. Для необходимо указать в аннотации второго параметра функционального компонента обобщенный тип взятого у второго параметра функционального типа `ForwardRefRenderFunction<T, P>` при помощи типа `Parameters<T>` предназначенного для получения массива с типами соответствующих параметрам функции. Поскольку интересующий нас тип принадлежит второму параметру, то он будет

доступен как элемент под индексом один. Кроме того, в указании аргументов типа универсальной функции `forwardRef<T, P>()` нет необходимости, поскольку выводу типов достаточно описания сигнатуры функционального компонента.

ts

```ts
import React, { forwardRef, ForwardRefRenderFunction } from "react";

export interface FormProps {}
export interface FormApi {
    reset: () => void;
}

  /**[0][1]       [2]             [3]            [4] [5] */
type Ref<T> = Parameters<ForwardRefRenderFunction<T>>[1];

                                  /**[6]   [7] */
function Form(props: FormProps, ref: Ref<FormApi>) {
    return null;
}

                         /**[8] */
const FormWithRef = forwardRef(Form);

export default FormWithRef;

/**
 * При помощи типа Parameters<T> [2] получаем массив элементы которого
 * принадлежат к типам параметров функции представляемой типом
 * ForwardRefRenderFunction<T, P> [3] которому в качестве первого
аргумента
 * типа [4] устанавливаем параметр обобщенного псевдонима [1]. Таким
образом
 * Ref<T> ссылается на первый элемент массива [5] содержащего тип
указанный в аннотации
 * второго параметра (ref). Определенный псевдоним указываем в аннотации
 * второго параметра функционального компонента [6] установив в
качестве аргумента
 * типа тип нативного dom элемента [7]. При таком сценарии нет
необходимости
 * конкретизировать типы при помощи аргументов типа универсальной
функции
 * forwardRef<T, P>() [8].
 */
```

Последнее на, что стоит обратить внимание, это обобщенный тип `ForwardRefExoticComponent<P>`, к которому принадлежит значение, возвращаемое из универсальной функции `forwardRef<T, P>()`, и который указывать в явной форме нет никакой необходимости.

ts

```
/**[0] */
import React, { forwardRef, ForwardRefRenderFunction,
ForwardRefExoticComponent } from "react";

// ...

                              /**[1]                [2] */
const FormWithRef: ForwardRefExoticComponent<FormProps> =
forwardRef(Form);

export default FormWithRef;

/**
 * [0] импорт обобщенного типа ForwardRefExoticComponent<P>
 * который в качестве аргумента типа ожидает тип представляющий пропсы
того
 * компонента [2] в аннотации типа которого указан [1]. Стоит заметить,
 * что указан он исключительно в образовательных целях.
 */
```

После того как функциональный компонент получит рефу, ему необходимо присвоить ей значение выступающее в качестве открытого *api*. Для этого необходимо прибегнуть к помощи хука `useImperativeHandle<T, R>(ref: Ref<T>, apiFactory() => R): void` подробное рассмотрение которого можно найти в теме посвященной предопределенным хукам.

Для того, что бы ассоциировать *api* компонента с компонентной рефой при помощи хука `useImperativeHandle()` , ему необходимо передать её в качестве первого аргумента. После этого, компонентная рефа будет ассоциирована со значением возвращаемого из функции ожидаемой хуком в качестве второго параметра. Процесс переинициализации компонентной рефы будет выполняться всякий раз при изменении элементов массива ожидаемого в качестве третьего параметра данного хука. Также необходимо уточнить, что рефа создаваемая с помощью хука `useRef()` и предназначенная для ассоциации с функциональным компонентом, также нуждается в явном преобразовании к обобщенному типу `MutableRefObject<T>` , которому в качестве единственного аргумента типа будет установлен тип представляющий открытое *api* компонента.

ts

```
import React, { forwardRef, ForwardRefRenderFunction, RefObject, useRef,
useImperativeHandle, MutableRefObject } from "react";

interface FormProps {}
interface FormApi {
    reset: () => void;
}

type Ref<T> = Parameters<ForwardRefRenderFunction<T>>[1];

function Form(props: FormProps, ref: Ref<FormApi>) {
                    /**[0] */
```

```
    let formRef = useRef() as RefObject<HTMLFormElement>;

        /**[1]          [2]     [3] */
    useImperativeHandle(ref, () => ({
        reset: () => formRef.current?.reset()
    }), [] /**[4] */);


    return <form ref={formRef}></form>;
}

/**
 * [0] не забываем о необходимости явного преобразования.
 * Хук useImperativeHandle ожидает в качестве первого параметра
 * ссылку [1], в качестве второго фабричную функцию [3], которая
 * будет переопределять объект api каждый раз при изменении
 * элементов массива ожидаемого в качестве третьего параметра [4].
 *
 */

const FormWithRef = forwardRef(Form);


const App = () => {
                        /**[5]         [6]        [7] */
    let formRef = useRef() as MutableRefObject<FormApi>;

    /**[8] */
    formRef.current?.reset();

                /**[9] */
    return <FormWithRef ref={formRef}/>;
}

/**
 * Необходимо помнить, что ссылка предназначенная
 * для ассоциации с функциональным компонентом
 * также требует явного преобразование [5] к обобщенному
 * типу MutableRefObject<T> [6] которому в качестве аргумента
 * типа необходимо установить тип представляющий открытое api
 * компонента [7]. И после создания экземпляра компонента определенного
 * с помощью функции forwardRef [9] можно использовать его api через
объект
 * рефы [8].
 */
```

В связи с тем, что функциональный компонент может определять только два параметра, пора перейти к рассмотрению следующего звена его сигнатуры — аннотации типа возвращаемого им значения. Для этого возвратимся к первоначальному, модифицированному с учетом пропсов, примеру функционального компонента у которого тип возвращаемого значения не указан явно, что в большинстве случаев является предпочтительней явного указания.

ts

```ts
import React from "react";

export interface TimerProps {}

export default function Timer(props: TimerProps) /**[0] */ {
    return <div>Is Timer!</div>;
}


/**
 * [0] отсутствие явного указания типа
 * возвращаемого типа, которая для большинства
 * случаев является предпочтительным.
 */
```

И дело не в том, что бы как можно больше делегировать работы выводу типов экономя тем самым драгоценное время, а в том, что система типов *React*, устанавливаемая из репозиториев `@types`, не имеет достаточно высокого уровня типобезопасности. Поскольку это очень щекотливая тема её освещение стоит начать с самого начала, а именно с перечисления типов к которым может принадлежать возвращаемое значение.

И так, любое допустимое возвращаемое компонентом значение, в системе типов *React*, может быть представлено типом `ReactNode` являющимся объединением (`Union`) определяемого типами `ReactChild | ReactFragment | ReactPortal | boolean | null | undefined`. Тип `ReactChild` также представляет собой объединение типов `ReactElement<Props, Type> | ReactText`. Первый, как уже было рассмотрено ранее, представляет экземпляр любого компонента и элемента *React*, а второй объединение `string | number`. `ReactFragment` представляет объединение для `{} | ReactNodeArray`. Не сложно догадаться, что `ReactNodeArray`, это абстракция над `Array<ReactNode>`. Оставшийся тип `ReactPortal` является производным от типа `ReactElement`. Это может казаться очень запутанным и более того разбираться в этом прямо сейчас нет нужды, поскольку совсем скоро станет ясно, в чем кроется подвох, причиной которого являются два из перечисленных типа.

Первый тип, вносящий смуту, это ранее рассмотренный `ReactElement<P, T>` и всё неожиданное поведение которое с ним связанно. Вторым типом вносящий неразбериху стал `ReactFragment`, поскольку определяющий его пустой объектный тип `{}` совместим с любым экземпляром объектного типа. По факту, при использовании в качестве типа возвращаемого значения `ReactFragment` или `ReactNode` ошибки не возникнет даже если оно будет экземпляром `Promise` или чего-то ещё. И хотя отсутствие ошибки на этапе компиляции не означает, что её получится избежать во время выполнения, сам сценарий с возвратом ошибочного значения может показаться чересчур надуманным. С какой-то долей вероятности можно с этим согласится, но поскольку идеология *TypeScript* подразумевает выявление проблем в программах до их запуска, об этом стоило хотя бы упомянуть.

ts

```
import React, { ReactFragment, ReactNode } from "react";


function A(): ReactFragment {
    return Promise.resolve(0); /**[0] */
}
function B(): ReactNode {
    return Promise.resolve(0); /**[0] */
}

/**
 * [0] Ok на этапе компиляции и
 * Error во время выполнения.
 */


/**[1] */
class T {
    constructor(readonly p: number){}
}

function C(): ReactFragment {
    return new T(0); /**[2] */
}
function D(): ReactNode {
    return new T(0); /**[2] */
}

/**
 * [1] определение некоторого класса.
 * [2] Ok на этапе компиляции и
 * Error во время выполнения.
 */
```

Из всего этого следует, что прибегать к аннотации типа возвращаемого значения стоит только в случаях когда оно принадлежит к `number`, `string`, `boolean`, `null` или массиву элементы которого принадлежат к оному из четырех перечисленных типов. Да и, то при острой необходимости. В остальных случаях целесообразней возложить эту работу на вывод типов, для которого это обычное дело.

Последнее, что осталось без внимания, это событийный механизм или если быть точнее определение слушателей событий. Для этого в системе типов *React* определен специальный обобщенный тип `ReactEventHandler<T>` ожидающий в качестве аргумента типа тип представляющий нативный *dom элемент*, которому будет установлен текущий слушатель событий.

Представим сценарий в котором по нажатию на кнопку с типом `submit` необходимо вернуть первоначальные значения элементов формы. Для этого потребуется подписать элемент формы на событие `submit` и по его возникновению вызвать у нативного элемента формы, ссылка на которую доступна через свойство событийного объекта `target`, метод `reset`.

Первым делом реализация подобного сценария потребует импорта обобщенного типа `ReactEventHandler<T>`, который в качестве аргумента типа получит тип нативного *dom элемента* `HTMLFormElement`, после чего будет указан в аннотации слушателя событий `form_submitHandler`. Выбор типа нативного *dom элемента*, в данном случае `HTMLFormElement`, обуславливается типом элемента, которому устанавливается слушатель событий, в данном случае `<form>`.

Стоит также обратить внимание, что единственный параметр слушателя событий в аннотации типа не нуждается, поскольку вывод типов опирается на обобщенный тип `ReactEvenHandler<T>`.

По возникновению события, первым делом необходимо предотвратить поведение по умолчанию, что бы избежать перезагрузки вкладки браузера. Поскольку ссылка на нативную форму доступна через определенное в объекте события свойство `target`, которое принадлежит к типу `EventTarget`, то перед присвоением её переменной `form` появляется необходимость в приведении к типу `HTMLFormElement` с помощью оператора `as`. После это можно вызывать нативный метод `reset`.

ts

```ts
/**[0] */
import React, {ReactEventHandler} from "react";

function Form(){
            /**[1]                  [2]              [3]           [4] */
    const form_submitHandler: ReactEventHandler<HTMLFormElement> =
event => {
        event.preventDefault(); /**[5] */

        /**  [6]        [7]               [8] */
        let form = event.target as HTMLFormElement;
        form.reset(); /**[9] */
    };


    return (
                           /**[10] */
        <form onSubmit={form_submitHandler}>
            <button type="submit">submit</button>
        </form>
    );
}

/**
 * [0] импорт обобщенного функционального типа
 * которому установив в качестве аргумента типа
 * тип нативного элемента HTMLFormElement [3]
 * использовали в аннотации типа [2] переменной
 * form_submitHandler [1], которой в качестве
 * значения присвоили функцию слушатель события
 * единственный параметр которой не нуждается в
 * явной аннотации типа [4], поскольку вывод типов
```

```
 * опирается на тип ReactEventHandler<T>.
 *
 * При возникновении события первым делом происходит
 * предотвращение поведения по умолчанию, что бы избежать
 * перезагрузки вкладки браузера [5]. Затем создается
 * переменная form [6] которой присваивается ссылка на
 * нативный dom элемент доступный через свойство определенное
 * в объекте события target [7] которое при помощи оператора
 * as приведено к нужному типу нативного dom элемента HTMLFormElement
[8].
 *
 * На следующем шаге вызывается нативный метод сброса значений формы
reset [9]
 *
 * [10] установка слушателя событий React элементу form.
 *
 */
```

Несмотря на то, что такой способ типизирования слушателей событий является предпочтительным, также не будет лишним рассмотреть и другой имеющийся вариант состоящий в описании непосредственно сигнатуры функции.

Для этого, в нашем конкретном случае, необходимо импортировать обобщенный тип `FormEvent<T>` , которому перед размещением в аннотации единственного параметра слушателя события необходимо в качестве аргумента события указать тип нативного *dom элемента* `HTMLFormElement` . Также стоит напомнить, что в аннотации возвращаемого из слушателя события значения нет необходимости. Подобную рутинную работу необходимо делегировать выводу типов.

ts

```
/**[0] */
import React, {FormEvent} from "react";

function Form(){
                                /**[1]      [2]           [3]
[4] */
    const form_submitHandler = (event: FormEvent<HTMLFormElement>): void
=> {

    }


    return (
        <form onSubmit={form_submitHandler}>
            <button type="submit">submit</button>
        </form>
    );
}

/**
 * [0] импорт обобщенного типа FormEvent<T>
```

```
 * которому перед добавлением в аннотацию типа [2]
 * единственного параметра слушателя события [1]
 * необходимо установить в качестве аргумента типа
 * тип нативного dom элемента HTMLFormElement [3].
 * Указании типа к которому принадлежит возвращаемое
 * из слушателя события значения было указанно лишь
 * для того, что бы напомнить об отсутствии в этом необходимости.
 * Подобную работу нужно делегировать выводу типов.
 */
```

Работа непосредственно с формой обусловила выбор более конкретного типа события, каковым в данном примере стал обобщенный тип `FormEvent<T>`. При других условиях потребуются другие событийные типы. Кроме того, всегда можно сделать выбор в пользу базового для всех событийных типов `SyntheticEvent<T>` ожидающего в качестве аргумента типа тип нативного *dom элемента*.

Кроме этого, функциональным компонентам доступна мемоизация слушателей событий при помощи универсального хука `useCallback<T>()`. Для этого понадобится импортировать универсальную функцию определяющую два обязательных параметра. В качестве первого параметра ожидается функция чье описание устанавливается в качестве аргумента функционального типа. Второй параметр принадлежит к типу массива, изменение элементов которого приводит к переинициализации функции переданной в качестве первого параметра. Поскольку в качестве аргумента функционального тпа ожидается тип описывающий первый параметр хука, то нет необходимости в аннотациях типа её параметров. Или в данном случае её единственного параметра представляющего объект события. В остальном же реализация ничем не отличается от предыдущего примера, поэтому повторяющийся код будет исключён.

ts

```ts
/**[0] */
import React, {useCallback, ReactEventHandler} from "react";

function Form(){
                                /**[1]
[2]                        [3]*/
    const form_submitHandler =
useCallback<ReactEventHandler<HTMLFormElement>>(event => {

    }, [] /**[4] */ );


    return (
        <form onSubmit={form_submitHandler}>
            <button type="submit">submit</button>
        </form>
    );
}

/**
 * [0] импорт универсальной функции useCallback<T>()
```

```
 * принимающей в качестве первого обязательного параметра
 * функцию [3], описание которой устанавливается в качестве
 * аргумента функционального типа [1]. D качестве второго
 * обязательного параметра ожидается массив [4] со значениями
 * изменение которых приводит переинициализации функции переданной
 * в качестве первого аргумента.
 */
```

На этом рассмотрение типизирования функционального компонента определенного как *Function Declaration*, завершено. И поскольку тема получилась довольно не маленькая исключим затягивание и перейдем к рассмотрению следующего вида функциональных компонентов.

[51.1] Определение компонента как Function Expression

Поскольку самое необходимое относящиеся ко всем видам *React* компонентов было рассмотрено в предыдущей теме, в этой и последующих, повествование будет сосредоточено исключительно на различиях.

Для определения функционального компонента как *Function Expression* декларация типов *React* предусматривает вспомогательный обобщенный тип `FC<Props>`, чей идентификатор (имя) является сокращением от *Function Component*, а аргумент типа представляет пропсы и является необязательным. Поскольку вывод типов ориентируется на тип пропсов указанный или присущий по умолчанию в качестве аргумента типа, то в аннотировании первого параметра функционального компонента нет надобности. Помимо этого, тип пропсов, по умолчанию описывает необязательное поле `children` принадлежащего к типу `ReactNode`.

ts

```ts
/**[0] */
import React, {FC} from "react";

          /**[1]       [2] */
const Timer: FC = ({children}) => <div>Is Timer!</div>;

/**
 * [0] импорт обобщенного типа FC<P>
 * который указан в аннотации без
 * установки аргумента типа [1] и
 * несмотря на это ошибки при деструктуризации
```

```
 * поля children не возникает даже без аннотации
 * типа первого параметра [2]
 */
```

Если тип пропсов указан в качестве аргумента типа `FC<P>` и при этом не описывает поле `children`, то оно всё равно будет определенно в объекте пропсов доступного в качестве первого параметра функционального компонента.

ts

```ts
import React, {FC} from "react";

export interface TimerProps {
    duration: number;
    /**[0] */
}
                 /**[1]                       [3]*/
const Timer: FC<TimerProps> = ({duration, children}) => <div>Is Timer!</
div>;


export default Timer;

/**
 * [0] несмотря на то, что тип представляющий
 * пропсы и указанный в качестве аргумента
 * типа FC<P> [1] не описывает поле children
 * при их деструктуризации ошибки не возникает [3]
 */
```

В остальном все, что было рассмотрено в предыдущей теме относительно пропсов и `children` идентично и для данного способа типизирования функциональных компонентов.

При возникновении необходимости в определении второго параметра функционального компонента придется самостоятельно указывать аннотацию типа, поскольку по каким-то причинам обобщенный тип `FC<P>` этого не предусматривает.

ts

```ts
import React, { FC } from "react";

export interface TimerProps {}


                            /**[1]   [2] */
const Timer: FC<TimerProps> = (props, ref) => <div>Is Timer!</div>;

/**
 * Первый параметр [0] выводится как
```

```
 * PropsWithChildren<TimerProps>, а
 * второй [2] как any, поскольку обобщенный
 * тип FC<P> не предусматривает его наличие.
 */
```

В остальном все рассмотренное относительно рефов в теме посвященной функциональным компонентам определенным как *Function Declaration*, верно и для текущего вида определения.

При необходимости во втором параметре можно отказаться от типа `FC<P>` в пользу ранее рассмотренного типа `ForwardRefRenderFunction<T, P>`. При указании данного типа в аннотации функционального компонента, пропадает необходимость, как в явном аннотировании типов его параметров, так и в указании аргументов типа универсальной функции `forwardRef<T, P>()`.

ts

```ts
import React, { FC, forwardRef, ForwardRefRenderFunction } from "react";

export interface TimerProps {}


                    /**[1] */
const Timer: ForwardRefRenderFunction<HTMLDivElement, TimerProps> =
(props, ref) => <div>Is Timer!</div>;


                        /**[2] */
const TimerWithRef = forwardRef(Timer);

export default TimerWithRef;

/**
 * [0] импорт типа для указания его в аннотации
 * функционального компонента [1] определяющего второй
 * параметр ref. После этого нет необходимости в явной
 * аннотации типов как обоих параметров функционального
 * компонента, так и универсальной функции forwardRef<T, P>().
 */
```

Важной особенностью использования обобщенного типа `FC<P>` заключается в том, что он, помимо типа представляющего пропсы, также содержит описание типа возвращаемого функцией значения. Вроде бы так и должно быть, но нюанс заключается в том, что возвращаемое значение обязательно должно принадлежать к типу совместимому с `ReactElement<P, T>`. Простыми словами на этапе компиляции возникнет ошибка если функциональный компонент, определенный как *Function Expression*, будет возвращать значение принадлежащие к типам `number`, `string`, `boolean` или абсолютно любому массиву.

ts

```
import React, {FC} from "react";

const A: FC = () => 0123; // Error
const B: FC = () => "0123"; // Error
const C: FC = () => true; // Error
const D: FC = () => []; // Error

const E: FC = () => <div></div>; // Ok
const F: FC = () => <E/>; // Ok
const G: FC = () => <></>; // Ok
```

Поэтому в случае, предполагающем, что функциональный компонент, определенный как *Function Expression*, будет возвращать значение отличное от `ReactElement<P, T>`, потребуется самостоятельно описать его сигнатуру. Что не представляет никакого труда.

ts

```
import React,{MutableRefObject} from "react";

export interface TimerProps {}

const Timer = (props: TimerProps, ref: MutableRefObject<HTMLDivElement>)
=> 123;
```

Если в подобном определении функциональных компонентов существует частая потребность, будет целесообразней определить собственный тип подробно описывающий сигнатуру функции.

Для этого потребуется определить обобщенный тип с двумя необязательными параметрами. Первый необязательный параметр представляющий тип пропсов должен расширять и к тому же указывать в качестве типа по умолчанию тип `object`. Второй необязательный параметр типа должен проделать тот же процесс только для нативного типа `HTMLElement`.

Чтобы не заморачиваться в определении `children`, указываем принадлежность первого параметра функции к уже знакомому обобщенному типу `PropsWithChildren<P>`, которому в качестве аргумента типа устанавливаем первый параметр типа `P`. Второму необязательному параметру функции указываем принадлежность к обобщенному типу `MutableRefObject<E>` в качестве аргумента типа которому устанавливаем второй параметр типа `E`. Осталось лишь указать принадлежность возвращаемого функцией значения к типу `ReactNode` и тип `CFC<P, E>`, что является сокращением от *Custom Functional Component*, готов сэкономить время и нервы разработчика.

ts

```
// file CFC.ts

import React,{MutableRefObject, ReactNode, PropsWithChildren} from
```

```
"react";

        /**[0][1]         [2]     [3][4]           [5]
[6]                     [7]                    [8]             [9]*/
export type CFC<P extends object = object, E extends HTMLElement =
HTMLElement> = (props: PropsWithChildren<P>, ref?: MutableRefObject<E>)
=> ReactNode;

/**
 * [0] определяем обобщенный тип CustomFunctionComponent
 * или сокращенно CFC первый необязательный параметр представляющего
 * пропсы которого [1] расширяет [2] и устанавливает по умолчанию [3]
 * тип object. Второй необязательный параметр представляющий тип
нативного
 * dom элемента [4] расширяет [5] и устанавливает по умолчанию [6] тип
HTMLElement.
 *
 * [7] устанавливаем принадлежность первого параметра функционального
типа к
 * обобщенному типу PropsWithChildren<P> которому в качестве аргумента
типа передаем первый
 * параметр типа.
 *
 * [8] определяем принадлежность второго необязательного параметра к
обобщенному типу
 * MutableRefObject<E> которому в качестве аргумента типа устанавливаем
второй параметр типа.
 *
 * [9] тип возвращаемого значения определяем как ReactNode.
 *
 */


// file Timer.tsx
import React from "react";
import {CFC} from "./CFC"; /**[0] */


export interface TimerProps {}

          /**[1]    [2]           [3]             [4]   [5]     [6]*/
const Timer: CFC<TimerProps, HTMLDivElement> = (props, ref) => 123;

/**
 * [0] импорт CustomFunctionComponent для
 * указания его в качестве типа функционального
 * компонента определенного как Function Expression [1].
 * В качестве первого параметра типа устанавливается тип
 * представляющий пропсы [2], а в качестве второго тип нативного
 * dom элемента с которым будет ассоциирован объект реф [3].
 * При таком подходе отпадает необходимость в явном указании аннотации
 * типов как пропсов [4], так и рефы [5]. Кроме того, возвращаемое
значение
```

```
 * может принадлежать к любому типу совместимому с типом ReactNode [6]
 */
```

На этом тема относящаяся функциональных компонентов себя полностью исчерпала, поэтому без лишних слов двигаемся к следующей теме посвященной классовым компонентам.

Глава 52

Классовые компоненты

Помимо компонентов на основе функций, *React* позволяет определять компоненты на основе классов, которые, как в случае реализации *ловушки для ошибок* просто не заменимы. Кроме этого, с ними необходимо познакомиться, так как они являются частью *React*. Именно поэтому текущая глава полностью посвящена старым добрым классовым компонентам.

[52.0] Производные от Component<P, S, SS>

Пользовательские компоненты построенные на основе классов обязаны расширять базовый обобщенный класс `Component<Props, State, Snapshot>` имеющего три необязательных параметра типа.

ts

```ts
import React, {Component} from "react";


class Timer extends Component {
    render(){
        return null;
    }
}

export default Timer;
```

Первым делом стоит обратить внимание на первую строку, а именно импорт пространства имен *React*. Не зависимо используете вы его напрямую или нет, оно обязательно должно быть импортировано, в противном случаи компилятор напомнит об этом с помощью ошибки.

ts

```ts
/**
 * [0] Забыт импорт пространства
 * имен React в следствии чего в
 * точке [1] возникнет ошибка -
 *
 * 'React' refers to a UMD global,
 * but the current file is a module.
 * Consider adding an import instead.ts(2686)
 */

import {Component} from "react"; // [0]


class Timer extends Component {
    render(){
        return null; // [1]
    }
}

export default Timer;
```

Кроме того, в нашем примере у метода `render` отсутствует аннотация возвращаемого типа, что на практике даже приветствуется. Но с образовательной точки зрения её указание не принесет никакого вреда.

ts

```ts
import React, {Component, ReactNode} from "react";


class Timer extends Component {
    render(): ReactNode {
        return null;
    }
}

export default Timer;
```

При переопределении производным классом метода `render` в качестве типа возвращаемого значения необходимо указывать тип совместимый с указанным в базовом классе, то есть с типом `ReactNode` поведение и нюансы которого были подробно рассмотрены в главе посвященной функциональным компонентам.

Как говорилось ранее, тип от которого должны наследоваться пользовательские классовые компоненты является обобщенным и имеет три необязательных параметра типа, что и иллюстрирует наш минималистический пример.

ts

```ts
/**
 * [0] отсутствует передача аргументов типа
 * определенных как Component<Props, State, Snapshot>
 *, что указывает на их необязательность.
 */
class Timer extends Component /** [0] */ {

}
```

В реальных проектах подобное встречается редко, поэтому следующим шагом разберем логику определения типов описывающих пользовательский компонент.

Начнем по порядку, а именно с `Props`. Несмотря на то, что *пропсы* делятся на обязательные и необязательные, все они по мере необходимости передаются в качестве аргументов конструктора при создании его экземпляра и доступны по ссылке `this.props` (обозначим их как *общие пропсы*). Тем не менее за инициализацию необязательных пропсов ответственен сам классовый компонент для чего и предусмотренно статическое поле `defaultProps`.

ts

```ts
/**
 * Аннотации в ожидании указания
 * типа.
 */
class Timer extends Component {
    public static readonly defaultProps /** [0] */ = {};

    constructor(props /** [1] */){
        super(props);
    }
}
```

Тот факт, что аннотация `defaultProps` предполагает тип представляющий лишь ассоциированное с этим полем значение вынуждает разделить декларацию общих пропсов на два типа `DefaultProps` и `Props`. Ввиду того, что тип `Props` представляет не только обязательные пропсы, но и необязательные, он должен расширять (`extends`) тип `DefaultProps`.

ts

```ts
interface DefaultProps {}
interface Props extends DefaultProps {}
```

```
class Timer extends Component {
    public static readonly defaultProps = {};

    constructor(props: Props){
        super(props);
    }
}
```

Не будет лишним упомянуть, что в реальных проектах интерфейс `Props`, помимо `DefaultProps`, очень часто расширяет множество других интерфейсов. В их число входят типы, предоставляемые библиотеками *ui*, *hoc обертками* и обычными библиотеками, как например *react-router* и его тип `RouteComponentProps<T>`.

Поскольку в описании базового класса поле (`this.props`) принадлежит к типу определенного в качестве первого параметра типа, то есть `Component<Props>`, то `Props` необходимо указать в аннотации не только первого параметра конструктора, но и в качестве первого аргумента базового типа. Иначе `this.props` так и останется принадлежать к простому объектному типу `{}` заданному по умолчанию.

ts

```
interface DefaultProps {
    message: string;
}
interface Props extends DefaultProps {
    duration: number;
}

/**
 * Если не передавать Props в качестве
 * аргумента типа в точке [0], то в точке
 * [1] возникнет ошибка ->
 * Property 'message' does not exist on type
 * 'Readonly<{}> & Readonly<{ children?: ReactNode; }>'
 */
class Timer extends Component<Props /**[0] */> {
    public static readonly defaultProps = {
        message: `Done!`
    };

    constructor(props: Props){
        super(props);

        props.message; // Ok
        this.props.message; // Ok [1]
    }
}
```

Как было сказано в теме посвященной функциональным компонентам, что если взять за правило именовать типы пропсов как `DefaultProps` и `Props`, то при необходимости в их импорте непременно возникнет коллизия из-за одинаковых имен. Поэтому принято

добавлять к названиям названия самих компонентов `*DefaultProps` и `*Props`. Но поскольку эти типы повсеместно указываются в аннотациях расположенных в теле классового компонента, то подобные имена попросту усложняют понимание кода. Поэтому для исчерпывающих имен необходимо создавать более компактные псевдонимы типа `type`.

Также стоит сразу сказать, что все три типа выступающих в качестве аргументов базового типа нуждаются в более компактных идентификаторах определяемых с помощью псевдонимов. Но, кроме того, все они описывают объекты, мутация которых не предполагается. Простыми словами типы `Props`, `State` и `Snapshot` используются исключительно в аннотациях `readonly` полей класса, параметрах его методов и возвращаемых ими значениях. Поскольку секрет здорового приложения кроется в типобезопасности, всю упомянутую троицу необходимо сделать неизменяемой. Для этого существует специальный тип `Readonly<T>`. Но так как преобразование типов в каждой отдельной аннотации приведет к чрезмерному увеличению кода, необходимо проделать это единожды в определении их псевдонимов.

Посмотрим как новая информация преобразит наш основной пример.

ts

```ts
import React, {Component, ReactNode} from "react";

/**
 * Имена интерфейсов получили префикс
 * в виде названия компонента.
 */
interface TimerDefaultProps {
    message: string;
}
interface TimerProps extends TimerDefaultProps {
    duration: number;
}

/**
 * Для конкретных типов преобразованных
 * в типы только для чтения
 * определен псевдоним.
 */
type DefaultProps = Readonly<TimerDefaultProps>;
type Props = Readonly<TimerProps>;

class Timer extends Component<Props> {
    public static readonly defaultProps: DefaultProps = {
        message: `Done!`
    };

    constructor(props: Props){
        super(props);
    }
}
```

```
/**
 * Добавлен экспорт не только самого
 * компонента, но и типа представляющего
 * его основные пропсы.
 */
export default Timer;
export {TimerProps}; // экспортируем типа *Props
```

Также стоит упомянуть, что пропсы всех компонентов по умолчанию имеют определение необязательного (объявленного с модификатором `?:`) поля `children` принадлежащего к оговоренному ранее типу `ReactNode` . Простыми словами можно вообще не передавать аргументы базовому типу и компилятор не выдаст ошибку при обращении к полю `this.props.children` ;

ts

```
class Label extends Component {
    render(){
        return (
                    /**[0] */
            <h1>{this.props.children}</h1>
        );
    }
}

/**
 * [0] несмотря на то, что базовому
 * типу не были установлены аргумента типа
 * обращение к свойству children не вызывает
 * ошибки поскольку данное свойство определенно
 * в базовом типе.
 */

<Label>{"label"}</Label>; // string as children -> Ok [1]
<Label>{1000}</Label>; // number as children -> Ok [2]
<Label></Label>; // undefined as children -> Ok [3]

 /**
  * При создании экземпляров компонента Label
  * допустимо указывать в качестве children
  * как строку [1], так и числа [2] и кроме
  * того не указывать значения вовсе [3]
  */
```

В остальном `children` имеют, то же поведение и недостатки подробно описанные в главе посвященной функциональным компонентам. Поэтому оставим их и приступим к рассмотрению второго параметра базового типа `Component` , а именно к типу представляющего состояние компонента `Component<Props, State>` .

Несмотря на то, что состояние является закрытым от внешнего мира, тип представляющий его также принято называть с префиксом в роли которого выступает

название самого компонента. Причина кроется не только в соблюдении общего стиля кода относительно именования типов пропсов. На практике могут возникнуть коллизии имен при создании вложенных классовых компонентов, что является обычным делом при создании *hoc*. Поэтому для типа описывающего состояние компонента так же необходимо определить ещё и псевдоним и не забыть передать его в качестве второго аргумента базового типа и указать в аннотации поля `state`.

ts

```ts
import React, {Component, ReactNode} from "react";

interface TimerDefaultProps {
    message: string;
}
interface TimerProps extends TimerDefaultProps {
    duration: number;
}

// определение State
interface TimerState {
    time: number;
}

type DefaultProps = Readonly<TimerDefaultProps>;
type Props = Readonly<TimerProps>;
type State = Readonly<TimerState>; // создание псевдонима для типа

/**
 * [0] передача псевдонима State
 * в качестве второго аргумента
 * базового типа.
 */
class Timer extends Component<Props, State /** [0] */> {
    public static readonly defaultProps: DefaultProps = {
        message: `DOne!`
    };

    // определение поля state
    public readonly state: State = {
        time: 0
    };

    constructor(props: Props){
        super(props);
    }
}

export default Timer;
export {TimerProps};
```

Пора обратить внимание на момент связанный с объявлением `defaultProps` и `state`, которым необходимо указывать (или не указывать вовсе) модификатор доступа `pu`

`blic` , так как к ним должен быть доступ извне. Кроме того, не будет лишним добавить этим полям модификатор `readonly` , который поможет избежать случайных изменений.

Говоря о состоянии нельзя обойти стороной такой метод как `setState` необходимый для его изменения, о котором известно, что в качестве аргумента он может принимать как непосредственно объект представляющий новое состояние, так и функцию возвращающую его. Но поскольку первый случай ничего, что нас могло бы заинтересовать, из себя не представляет, рассмотрен будет лишь второй вариант с функцией. Поэтому продолжим наш основной пример и внесем в него изменения касающиеся изменения состояния. Создадим скрытый метод `reset` который будет сбрасывать значение пройденного времени.

ts

```
interface TimerState {
    time: number;
}

type State = Readonly<TimerState>;


class Timer extends Component<Props, State> {
    public static readonly defaultProps: DefaultProps = {
        message: `DOne!`
    };

    public readonly state: State = {
        time: 0
    };

    constructor(props: Props){
        super(props);
    }

    // определение скрытого метода reset
    private reset(){
        /**
         * Вызываем метод setState с функцией
         * асинхронного изменения состояния
         * в качестве первого аргумента.
         */
        this.setState( (prevState: Readonly<State>, props:
Readonly<Props>) => {
            return {time: 0}; // возвращаем новое состояние
        } )
    }
}
```

Из того кода, что был добавлен в наш пример стоит обратить внимание на несколько моментов. Прежде всего это использование псевдонимов `Props` и `State` в аннотациях параметров функции переданной в метод `setState` . Обозначим её как `up`

`dater` . Как было сказано ранее, типы описывающие состояние и пропсы используются повсеместно в коде компонента. Кроме того, стоит сказать, что описание сигнатуры функции `updater` подобным образом излишне и имеет место быт лишь в образовательных целях. Достаточно просто определить необходимые параметры и вывод типов самостоятельно определит их принадлежность.

ts

```ts
class Timer extends Component<Props, State> {
    private reset(){
        /**
         * Вывод типов в состоянии определить
         * принадлежность параметров, поэтому
         * самостоятельное аннотирование излишне.
         *
         * (parameter) prevState: Readonly<TimerState>
         * (parameter) props: Readonly<TimerProps>
         */
        this.setState( (prevState, props) => {
            return {time: 0};
        } )
    }
}
```

В добавок к этому стоит возложить определение возвращаемого значения из функции `updater` на вывод типов, поскольку это не просто излишне, но и в большинстве случаев может являться причиной избыточного кода. Все дело в том, что когда состояние содержит множество полей, обновление которых не производится одновременно, при указании возвращаемого типа как `State` будет невозможно частичное обновление, поскольку лишь часть типа `State` не совместимо с целым `State` .

ts

```ts
interface Props{}
interface State{ /**[0] */
  yesCount: number;
  noCount: number;
}
class Counter extends Component<Props,State>{
  state = {
    yesCount:0,
    noCount:0
  }

  buttonA_clickHandler = () => {
      // инкрементируем yesCount
    this.setState((prevState): State => {
      return {yesCount: prevState.yesCount + 1}; /**1 */
    });
  };
  buttonB_clickHandler = () => {
      // инкрементируем noCount
```

```
    this.setState((prevState): State => {
        return {noCount: prevState.noCount + 1}; /**[2] */
    });
  };


  render(){
    return (
      <div>
          <p>Yes: {this.state.yesCount}</p>
          <p>No: {this.state.noCount}</p>
          <button onClick={this.buttonA_clickHandler}>yes++</button>
          <button onClick={this.buttonB_clickHandler}>no++</button>
      </div>
    );
  }
}

/**
 * [0] описание состояния с двумя полями.
 * [1] Error -> поскольку {yesCount: number} не совместим
 * с {yesCount: Number; noCount: number}
 * [2] Error -> поскольку {noCount: number} не совместим
 * с {yesCount: Number; noCount: number}
 */
```

В случае когда функция `updater` выполняет частичное обновление состояния и при этом тип возвращаемого значения указан явно, необходимо воспользоваться механизмом распространения (`spread`) дополнив отсутствующую часть в новом состоянии старым.

ts

```
class Counter extends Component<Props,State>{
  buttonA_clickHandler = () => {
    this.setState((prevState): State => {
                /**[0] */
      return {...prevState, yesCount: prevState.yesCount + 1};
    });
  };
  buttonB_clickHandler = () => {
    this.setState((prevState): State => {
                    /**[1] */
        return {...prevState, noCount: prevState.noCount + 1};
    });
  };
}

/**
 * [0] В обоих случаях ошибки не возникает
 * поскольку недостающая часть состояния
 * дополняется из предыдущего состояния,
```

```
 *, что делает тип возвращаемого объекта
 * совместимым с типом State.
 */
```

Несмотря на то, что механизм распространения помогает обойти трудности связанные с совместимостью типов, лучшим вариантом будет вообще не указывать возвращаемый функцией `updater` тип, а возложить эту обязанность на вывод типов.

И последнее о чем ещё не упомянули, что метод `setState`, в качестве второго параметра принимает функцию обратного вызова, декларация которой очень проста и будет рассмотрена в самом конце данной главы, когда весь код будет собран в одном месте.

И на этом рассмотрение состояния завершено, поэтому можно приступить к рассмотрению третьего и последнего параметра базового типа `Component<Props, State, Snapshot>`.

Принципы применяемые для описания типа представляющего `Snapshot` ничем не отличаются от описания `Props` и `State`, поэтому пояснения будут опущены.

ts

```ts
import React, {Component, ReactNode} from "react";

interface TimerDefaultProps {
    message: string;
}
interface TimerProps extends TimerDefaultProps {
    duration: number;
}

interface TimerState {
    time: number;
}

// определение Snapshot
interface TimerSnapshot {}


type DefaultProps = Readonly<TimerDefaultProps>;
type Props = Readonly<TimerProps>;
type State = Readonly<TimerState>;
type Snapshot = Readonly<TimerSnapshot>; // создание псевдонима для
типа

/**
 * [0] передача псевдонима Snapshot
 * в качестве третьего аргумента
 * базового типа.
 */
class Timer extends Component<Props, State, Snapshot /** [0] */> {
    /**
```

```
     * Поскольку Snapshot используется
     * в тех конструкциях очередь до которых
     * ещё не дошла, тело класса будет опущено.
     */
}

export default Timer;
export {TimerProps};
```

Ничего особенного на, что стоило бы обратить внимание нет. Поэтому без лишних комментариев продолжим знакомство с внутренним устройством компонента — его жизненного цикла.

Погружение в типизированный жизненный цикл классовых компонентов необходимо начать с его разделения на две части — *актуальный жизненный цикл* и *устаревший жизненный цикл*, который будет исключён из рассмотрения. Поскольку в аннотации методов жизненного цикла не содержится ничего, что было бы непонятно к этому моменту, пояснение каждого отдельного случая будет опущено. Обратить внимание стоит лишь на импорт впервые встречающегося типа `ErrorInfo` необходимость в котором появляется при определении необязательно метода `componentDidCatch`. Кроме того, не будет лишнем напомнить, что в строгом, рекомендуемом режиме, при котором все элементы без аннотации неявно принадлежат к типу `any`, аннотация сигнатур методов является обязательной. И по этому случаю ещё раз стоит упомянуть о пользе коротких псевдонимов заменяющих огромные идентификаторы типов `*Props`, `*State` и `*Snapshot`.

ts

```
import React, {Component, ReactNode, ErrorInfo} from "react"; //
необходимость в импорте типа ErrorInfo

class Timer extends Component<Props, State, Snapshot> {
    getDerivedStateFromProps?:(nextProps: Readonly<Props>, prevState:
State) => Partial<State> | null;
    getDerivedStateFromError?: (error: any) => Partial<State> | null;

    componentDidMount?(): void
    shouldComponentUpdate?(nextProps: Readonly<Props>, nextState:
Readonly<State>, nextContext: any): boolean;
    componentWillUnmount?(): void;
    componentDidCatch?(error: Error, errorInfo: ErrorInfo): void;
    getSnapshotBeforeUpdate?(prevProps: Readonly<Props>, prevState:
Readonly<State>): Snapshot | null;
    componentDidUpdate?(prevProps: Readonly<Props>, prevState:
Readonly<State>, snapshot?: Snapshot): void;

}
```

Вдобавок необходимо заметить, что код иллюстрирующий жизненный цикл компонента взят из декларации, устанавливаемой из репозитория `@types/react`, и именно

поэтому она изобилует излишними преобразованиями в `Readonly<T>` тип. Но как было отмечено ранее, в этом нет нужны поскольку все типы составляющие троицу аргументов базового типа уже прошли преобразование при определении представляющих их псевдонимов. Учитывая этот факт предыдущий код будет выглядеть следующим образом.

ts

```
/**
 * Более компактная запись
 * без изменения поведения.
 */
class Timer extends Component<Props, State, Snapshot> {
    getDerivedStateFromProps?:(nextProps: Props, prevState: State) =>
Partial<State> | null;
    getDerivedStateFromError?: (error: any) => Partial<State> | null;

    componentDidMount?(): void
    shouldComponentUpdate?(nextProps: Props, nextState: State,
nextContext: any): boolean;
    componentWillUnmount?(): void;
    componentDidCatch?(error: Error, errorInfo: ErrorInfo): void;
    getSnapshotBeforeUpdate?(prevProps: Props, prevState: State):
Snapshot | null;
    componentDidUpdate?(prevProps: Props, prevState: State, snapshot?:
Snapshot): void;

}
```

Следующий в очереди на рассмотрение механизм, получение ссылок на нативные *dom элементы* и *React компоненты*, обозначаемый как *рефы* (*refs*).

Предположим, что существует форма, которую по событию `submit` необходимо очистить при помощи нативного метода `reset`, доступного лишь через нативный *dom элемент*, ссылку на который можно получить с помощью механизма рефов, применение которого возможно осуществить двумя способами. Первый способ заключается в создании объекта реф с помощью статического метода `React.createRef()`, а второй в самостоятельном сохранении ссылки на нативный *dom элемент* с помощью функции обратного вызова.

ts

```
/**
 * задача заключается в
 * получении ссылки на
 * нативный dom елемент формы [0].
 */
class CheckList extends Component {
    render(){
        return (
            /**[0] */
            <form></form>
```

```
        );
    }
}
```

Начнем по порядку. Первым делом необходимо определить поле (в нашем случае это `formRef`) необходимое для сохранения объекта реф и желательно, что бы оно было закрытое (`private`) и только для чтения (`readonly`). В примере поле `formRef` определен вместе с аннотацией в которой указан импортированный тип `RefObject<T>`, где параметр типа принимает тип нативного *dom элемента*, в нашем случае `HTMLFormElement`. Но в конкретном примере аннотация излишня поскольку мы указали выводу типов принадлежность нативного *dom элемента* передав его в качестве аргумента типа функции `React.createRef<T>()`.

ts

```ts
import React, {Component, RefObject} from "react";

class CheckList extends Component {
                                        /**[1]
*/                                       /**[2] */
    private readonly formRef: RefObject<HTMLFormElement> =
React.createRef<HTMLFormElement>();

}

/**
 * [0] импорт типа RefObject<T>
 * который в аннотации [1] поля
 * formRef является излишним,
 * так как тип нативного dom элемента
 * был уточнен с помощь передачи его
 * в качестве аргумента типа функции [2]
 */
```

На следующим шаге устанавливаем объект реф *react элементу* `<form>` и определяем закрытый метод `reset` в котором происходит вызов метода `reset` нативной формы. Не будет лишним обратить внимание, что вызов непосредственно метода `reset` осуществляется при помощи *оператора опциональной последовательности* (`?.`). Сделано это по причине возможного отсутствия ссылки на нативный элемент.

ts

```ts
import React, {Component, RefObject} from "react";


class CheckList extends Component {
    private readonly formRef: RefObject<HTMLFormElement> =
React.createRef<HTMLFormElement>();

    /**[4] */
    private resetForm(){
```

```
                        /**[5] */
        this.formRef.current?.reset();
    }

    render(){
        return (
                        /**[3] */
            <form ref={this.formRef}></form>
        );
    }
}

/**
 * [3] установка рефа react элементу.
 * [4] определение закрытого метода.
 * [5] необходимость применения оператора
 * опциональной последовательности по причине
 * возможного отсутствия ссылки на нативный элемент.
 */
```

Второй способ получения ссылки на нативный элемент заключается в определении функции принимающей в качестве единственного параметра нативный *dom элемент*, сохранение ссылки на который перекладывается на разработчика.

Для иллюстрации сказанного повторим предыдущий пример. Первым делом импортируем обобщенный тип `RefCallback<T>` описывающий функцию и принимающий в качестве аргумента типа тип нативного *dom элемента*, который будет передан в функцию в качестве единственного аргумента. Затем определим поле `formNativeElement` с типом `union`, множество которого включат не только тип нативного элемента, но и `null`. Это необходимо поскольку при инициализации требуется установить значение принадлежащие к типу `null`. Это необходимо при активном флаге `--strictPropertyInitialization` входящим в группировку определяющую рекомендуемый строгий режим компилятора.

Следующим шагом происходит определение закрытого только для чтения поля `formRefCallback` которому в качестве значения присвоена стрелочная функция. Единственный параметр данной функции лишен аннотации тпа, поскольку вывод типов определит его как принадлежащего к переданному в качестве аргумента типа `RefCallback<T>`. В теле данной функции происходит присваивание её параметра полю `formNativeElement` определенному на предыдущем шаге.

ts

```
/**[0] */
import React, {Component, RefCallback} from "react";


class CheckList extends Component {
                /**[1] */                /**[2] */     /**[3] */
    private formNativeElement: HTMLFormElement | null = null;
                        /**[4] */                      /**[5] */      /
```

```
**[6] */                          /**[7] */
    private readonly formRefCallback: RefCallback<HTMLFormElement> =
element => this.formNativeElement = element;

}

/**
 * [0] импорт типа RefCallback<T> который в качестве аргумента
 * типа ожидает тип нативного элемента.
 * [1] определение поля formNativeElement
 * и присвоение ему значения null [3], что приводит
 * к необходимости объединенного типа включающего
 * тип null [2]. [4] определение поля formRefCallback
 * значением которого служит стрелочная функция принимающая в
 * качестве единственного параметра нативный элемент [6] который затем
 * присваивается полю formNativeElement [7]. Тип этого параметра
 * будет принадлежать к типу переданному в качестве аргумента типа
RefCallback<T> [5]
 *
 */
```

Стоит заметить, что, то же самое можно реализовать и без помощи типа импортированного `RefCallback<T>` . Для этого лишь потребуется самостоятельно добавить аннотацию типа для параметра функции обратного вызова.

ts

```
import React, {Component} from "react";


class CheckList extends Component {
    private formNativeElement: HTMLFormElement | null = null;
    private readonly formRefCallback = (element: HTMLFormElement) =>
this.formNativeElement = element;

}

/**
 * [0] определение поля formNativeElement
 * и присвоение ему значения null [2], что приводит
 * к необходимости объединенного типа включающего
 * тип null [1]. [3] определение поля formRefCallback
 * значением которого служит стрелочная функция в качестве
 * аргумента которая ожидает нативный элемент [4] который
 * затем присваивается полю formNativeElement [5]
 *
 */
```

Выбор того или иного способа зависит лишь от предпочтений самого разработчика.

Продолжим доведение примера до финального состояния и установим созданную в первом случае функцию обратного вызова *react элементу* `<form>` в качестве реф. Также определим уже известный метод `reset` в теле которого будет происходить вызов метода `reset` у нативного *dom элемента* ссылка на который будет сохранена в поле класса `formNativeElement` .

ts

```
import React, {Component, RefCallback} from "react";


class CheckList extends Component {
    private formNativeElement: HTMLFormElement | null = null;
    private readonly formRefCallback: RefCallback<HTMLFormElement> =
element => this.formNativeElement = element;


    /**[1] */
    private reset(){
                /**[2] */
        this.formNativeElement?.reset();
    }


    render(){
        return (
                            /**[0] */
            <form ref={this.formRefCallback}></form>
        );
    }
}

/**
 * [0] устанавливаем callback в качестве значения реф
 * после чего определяем метод reset [1] в теле которого
 * при помощи оператора опциональной последовательности
 * вызываем метод reset у нативного dom элемента сохранённого
 * в поле formNativeElement [2]
 *
 */
```

И раз уж тема дошла до рассмотрения рефов, то необходимо рассмотреть механизм получения с их помощью ссылки на классовый компонент.

Первым делом определим классовый компонент `Slider` реализующий два открытых метода предназначенных для перелистывания контента `prev` и `next` . Далее определим компонент `App` в теле которого определим рефу при помощи функции `crea teRef` , которой (рефе) в качестве аргумента типа передадим тип классового компонента `Slider` . Таким образом вывод типа определит рефу `sliderRef` , как принадлежащую к типу `RefObject<Slider>` . После этого в методе рендер создадим экземпляр компонента `Slider` и два *react элемента* `<button>` , в обработчиках

событий `click` которых происходит взаимодействие с компонентом `Slider` при помощи ссылки на него доступной через ассоциированную непосредственно с ним рефу.

ts

```
import React, {Component, createRef} from "react";


class Slider extends Component{
    public prev = () => {}; /**[0] */
    public next = () => {}; /**[1] */

}

class App extends Component {
                    /**[2]                  [3]*/
    private readonly sliderRef = createRef<Slider>();


    render(){
        return (
            <>
                <button onClick={() => this.sliderRef.current?.prev()}
>prev</button> { /**[4] */}
                <Slider ref={this.sliderRef} /> { /**[5] */}
                <button onClick={() => this.sliderRef.current?.next()}
>next</button> { /**[6] */}
            </>
        )
    }
}

/**
 * [0] псевдо компонент Slider реализует
 * два доступных метода перелистывания контента
 * назад [0] и вперед [1]. Псевдо компонент App
 * определяет рефу с помощью универсальной функции
 * createRef в качестве аргумента типа которой был
 * установлен тип компонента Slider. В методе render
 * происходит определение двух пользовательских кнопок
 * выполняющих перелистывание по событию click, в обработчиках
 * событий которых происходит вызов доступных методов prev [4]
 * и next [6] через рефу ассоциированную непосредственно с компонентом
[5]
 */
```

На этом рассмотрение работы с механизмом рефов в типизированном стиле завершено. Но до завершения знакомства с работой классового компонента в основе которого лежит `Component<Props, State, Snapshot>` осталась ещё одна тема, а именно работа с *React событиями*. Кроме того, её освещение будет являться альтернативным решением задачи получения доступа к нативному элементу. Простыми словами реализуем вызов метода `reset` у нативного *dom элемента* ссылку на который будет

получена из объекта события `submit`. Но поскольку данная тема была подробна рассмотрена в главе посвященной функциональным компонентам, здесь подробно будут освещены только моменты присущие исключительно классовым компонентам.

Первым делом возвратим предыдущий пример в первоначальное состояние и добавим кнопку для отправки формы.

ts

```ts
import React, {Component} from "react";

class Form extends Component {
    render(){
        return (
            <form>
                <button type="submit"></button>
            </form>
        );
    }
}
```

Далее нам потребуется определить закрытое поле только для чтения в качестве значения которого будет присвоена стрелочная функция способная сохранить контекст текущего экземпляра. В качестве типа данного поля укажем импортированный из пространства имен *React* ранее рассмотренный обобщенный тип `ReactEventHandler<T>`.

ts

```ts
/**[0] */
import React, {Component, ReactEventHandler} from "react";

class Form extends Component {
                              /**[1] */                              /**[2] */     /**[3] */
    private readonly form_submitHandler: ReactEventHandler<HTMLFormElement> = event => {

    }

    render(){
        return (
                                      /**[4] */
            <form onSubmit={this.form_submitHandler}>
                <button type="submit"></button>
            </form>
        );
    }
}

/**
 * [0] импорт типа ReactEventHandler<T>
```

```
 * представляющего слушателя события.
 * [1]  Определение закрытого неизменяемого
 * поля принадлежащего к функциональному
 * типу ReactEventHandler<T>. [2] тип нативного
 * dom элемента определенного стандартной
 * библиотекой. [3] единственный параметр
 * функции не нуждается в аннотации поскольку
 * вывод типа опирается на ReactEventHandler<T>.
 * [4] установка слушателя.
 */
```

Для завершения примера осталось всего-навсего написать логику слушателя события `submit`, которая также повторяет пример из главы посвященной функциональным компонентам и поэтому подробных комментариев не будет.

ts

```
class Form extends Component {
    private readonly form_submitHandler:
ReactEventHandler<FormEvent<HTMLFormElement>> = event => {
        event.preventDefault(); // [0]
        let form = event.target as HTMLFormElement; // [1]
        form.reset(); // [2]
    }
}

/**
 * [0] для предотвращения отправки формы
 * и перезагрузки страницы прерываем стандартное
 * поведение. [1] поскольку доступ к форме можно
 * получить через ссылку свойства target принадлежащего
 * к типу EventTarget, появляется необходимость в
 * приведении к типу HTMLFormElement с при помощи оператора as.
 * [2] вызываем метод reset.
 */
```

Данный способ типизирования слушателей событий является предпочтительным поскольку при таком подходе аннотация включает только два типа и, кроме того, стрелочная функция уберегает от неминуемой потери контекста. Случаи требующие определения слушателя как метода класса требуют другого подхода. Отличие заключается в том, что в аннотировании типа нуждается непосредственно параметр слушателя. Но поскольку *React* делегирует все нативные события, необходимо импортировать тип соответствующего события из его пространства имен. Для событий связанных с формами в *React* определен обобщенный тип `FormEvent<T>` ожидающий в качестве аргумента типа тип нативного элемента. И поскольку слушатель ничего не возвращает, то тип возвращаемого значения, явное указание которого излишне, определяется как `void`.

ts

```
/**[0] */
import React, {Component, FormEvent} from "react";

class Form extends Component {
    /**[1] */                          /**[2] */        /**[3] */
    form_submitHandler(event: FormEvent<HTMLFormElement>): void {
    }
}

/**
 * [0] импортируем тип FormEvent<T> после
 * чего определяем метод form_submitHandler
 * тип единственного параметра которого определен
 * как FormEvent<HTMLFormElement>, а возвращаемое
 * значение [3] которое указанно лишь для того, что бы
 * напомнить об отсутствии необходимости в его явном указании.
 */
```

Поскольку установка слушателя представляемого методом класса приведет к неминуемой потери контекста, прибегать к подобному объявлению стоит только при условии, что их тело лишено логики предполагающей обращение к членам через ссылку экземпляра `this` .

ts

```
class Form extends Component {
    form_submitHandler(event: FormEvent<HTMLFormElement>): void {
        /**
         * Здесь нельзя обращаться к this
         * поскольку контекст на текущий экземпляр
         * был утерян.
         */
    }

    render(){
        return (
            <form onSubmit={this.form_submitHandler}></form>
        );
    }

}
```

Контекст можно было бы сохранить прибегнув к методу `bind` или делегированию события непосредственно с помощью стрелочной функции определенной в месте установки слушателя, но зачем? Для `bind` потребуется определения дополнительного поля.

ts

```
class Form extends Component {
    // дополнительное поле
    private form_submitHandlerBinded: (event:
FormEvent<HTMLFormElement>) => void;

    constructor(props:Props){
        super(props);

        // лишняя инициализация
        this.form_submitHandlerBinded =
this.form_submitHandler.bind(this);
    }


    form_submitHandler(event: FormEvent<HTMLFormElement>): void {
        /**
         * Теперь здесь можно обращаться к this
         */
    }

    render(){
        return (
            // в качестве слушателя установлена функция связанная с
помощью bind
            <form onSubmit={this.form_submitHandlerBinded}></form>
        );
    }

}
```

Стрелочная функция будет пересоздаваться каждую отрисовку.

ts

```
class Form extends Component {
    form_submitHandler(event: FormEvent<HTMLFormElement>): void {
        /**
         * Теперь здесь можно обращаться к this
         */
    }

    render(){
        return (
            // пересоздание функции каждую отрисовку
            <form onSubmit={event => this.form_submitHandler(event)}></
form>
        );
    }

}
```

Кроме того, оба случая затрудняют понимание кода. Поэтому необходимо повторить, что использовать метод класса в качестве слушателя события стоит только при отсутствии необходимости в обращении через ссылку `this`. При возникновении именно такого случая не будет лишним уточнения способа выбора типа события. В приведенном примере это был `FormEvent<T>`, поскольку работа производилась с формой. Для других событий появится необходимость в других соответствующих типа, узнать которые можно с помощью подсказок вашей *ide*. Для чего всего-лишь необходимо навести курсор на определение слушателя события.

ts

```ts
class Clicker extends Component {
    render(){
        return (
                /**[0] */
            <div onClick={}></div>
        )
    }

}

/**
 * [0] при наведении курсором
 * на определение слушателя onClick
 * ide подсказывает тип как MouseEvent<HTMLDivElement>
 */
```

Также не забываем об упомянутом ранее базовом для всех событийных *React* типов обобщенном типе `SyntheticEvent<T>`, который в качестве аргумента ожидает тип представляющий нативный элемент.

На этом тему посвященную созданию классового компонента расширяющего `Component<Props, State, Snapshot>` можно заканчивать и переходить к следующей теме. Единственное, что точно не будет лишним, так это собрать весь пройденный материал в одном месте.

ts

```ts
import React, {Component, ReactNode, ReactEventHandler, RefObject,
SyntheticEvent, ErrorInfo} from "react";

interface GreeterDefaultProps {} // для декларации свойств по умолчанию
export interface GreeterProps extends GreeterDefaultProps {
    children: ReactNode | ReactNode[]; // указываем, что children могут
принадлежать к единичному типу или множеству составляющего тип ReactNode
} // для декларации обязательных свойств + экспорт интерфейса
interface GreeterState {} // для декларации состояния
interface GreeterSnapshot {} // для декларации снимка

// создаем псевдонимы для readonly типов представляющих...
type DefaultProps = Readonly<GreeterDefaultProps>; // ... статическое
поле defaultProps
```

```
type Props = Readonly<GreeterProps>; // ... поле props
type State = Readonly<GreeterState>; // ... поле state
type Snapshot = Readonly<GreeterSnapshot>; // ... параметр snapshot
определенный в нескольких методах жизненного цикла


export default class Greeter extends Component<Props, State, Snapshot> {
    public static readonly defaultProps: DefaultProps = {}; //
модификатор readonly от случайного изменения статического поля
defaultProps которое должно иметь модификатор доступа public


    // необязательные методы класса (статические методы)
    public static getDerivedStateFromProps?:(nextProps: Props,
prevState: State) => Partial<State> | null;
    public static getDerivedStateFromError?: (error: any) =>
Partial<State> | null;


    public readonly state: State = {}; // модификатор readonly от
случайного изменения поля state которое должно иметь модификатор
доступа public


    /** два различных способа получения ссылки на нативный dom элемент
*/
    // [0] при помощи контейнера
    private readonly formRef: RefObject<HTMLFormElement> =
React.createRef(); // создание объекта RefObject, с помощью которого
будет получена ссылка на dom элемент

    // [1] при помощи callback
    private textRef: HTMLSpanElement | null = null; // поле, в которое
будет сохранена ссылка на DOM-элемент
    private readonly textRefCallback = (element: HTMLSpanElement) =>
this.textRef = element; // определение функции обратного вызова для
установления ссылки на DOM-элемент



    constructor (props: Props) {
        super(props);
    }


    // методы жизненного цикла
    public componentDidMount?(): void
    public shouldComponentUpdate?(nextProps: Props, nextState: State,
nextContext: any): boolean;
    public componentWillUnmount?(): void;
    public componentDidCatch?(error: Error, errorInfo: ErrorInfo): void;
    public getSnapshotBeforeUpdate?(prevProps: Props, prevState: State):
Snapshot | null;
```

```
    public componentDidUpdate?(prevProps: Props, prevState: State,
snapshot?: Snapshot): void;


    /** два варианта определения слушателя событий */
    // слушатель событий определенный как поле
    private readonly form_submitHandler:
ReactEventHandler<HTMLFormElement> = event => {
        // изменение состояния
        this.setState((prevState: State, prevProps: Props) => {
            return {};
        });
    };
    // слушатель событий определенный как метод
    private submitButton_clickHandler(event:
SyntheticEvent<HTMLButtonElement>): void {

    }

    public render(): ReactNode {
        return (
            <form ref={this.formRef} onSubmit={this.form_submitHandler}>
                <span ref={this.textRefCallback}>Send form?</span>
                <button type="submit"
onClick={this.submitButton_clickHandler}>yes</button>
            </form>
        );
    }
}
```

[52.1] Производные от PureComponent<Props, State, Snapshot>

Помимо того, что пользовательские компоненты могут быть производными от универсального класс `Component<Props, State, Snapshot>`, они также могут использовать в качестве базового класса универсальный класс `PureComponent<Props, State, Snapshot>`. Но поскольку все, что было сказано относительно `Component` в ста процентах случаев верно и для `PureComponent`, который также ничего нового не привносит, то данная глава будет ограничена лишь кодом иллюстрирующим определение пользовательского компонента.

ts

```
import React, { PureComponent } from "react";

/**[*] */

export default class Greeter extends PureComponent<Props, State,
Snapshot> {
    /**[*] */
}

/**
 * [*] здесь предполагается логика
 * рассмотренная в главе, посвященной
 * производным от Component<P, S, SS>
 */
```

Тем кто только начал своё знакомство с классовыми компонентами с данной главы необходимо вернутся на шаг назад или даже более разумно в самое начало, поскольку именно там объясняется, что для полного понимания необходимо ознакомиться со всем материалом относящимся к *React*.

Глава 53

Универсальные компоненты

Подобно универсальным классам, синтаксис `.tsx`, позволяет определять *React* компоненты обобщенными. С этим связано несколько неочевидных моментов, каждый из которых будет рассмотрен в текущей главе.

[53.0] Обобщенные компоненты (Generics Component)

В *TypeScript* существует возможность объявлять пользовательские компоненты обобщенными, что лишь повышает их повторное использование. Чтобы избавить читателя от пересказа того, что подробно было рассмотрено в главе "Типы - Обобщения (Generics)", опустим основную теорию и сосредоточимся конкретно на той её части, которая сопряжена непосредственно с *React* компонентами. Но поскольку польза от универсальных компонентов может быть не совсем очевидна, прежде чем приступить к рассмотрению их синтаксиса, стоит упомянуть, что параметры типа предназначены по большей степени для аннотирования членов типа представляющего пропсы компонента.

В случае компонентов, расширяющих универсальные классы `Component<P, S, SS>` или `PureComponent<P, S, SS>`, нет ничего особенного, на, что стоит обратить особое внимание.

ts

```ts
/**[0] */
interface Props<T> {
```

```
    data: T; /**[1] */
}

  /**[2][3]                          [4] */
class A<T> extends Component<Props<T>> {}
  /**[2][3]                            [4] */
class B<T> extends PureComponent<Props<T>> {}


// ...где-то в коде



/**[5] */
interface IDataB {
    b: string;
}

/**[6] [7]             [8] */
   <A<IDataA> data={{a: 0}} />; // Ok
/**[6] [7]             [9] */
   <A<IDataA> data={{a: '0'}} />; // Error


/**[5] */
interface IDataA {
    a: number;
}

/**[6] [7]             [8] */
   <A<IDataB> data={{b: ''}} />; // Ok
/**[6] [7]             [9] */
   <A<IDataB> data={{b: 0}} />; // Error

/**
 * [0] определение обобщенного типа чей
 * единственный параметр предназначен для
 * указания в аннотации типа поля data [1].
 *
 * [2] определение универсальных классовых
 * компонентов чей единственный параметр типа [3]
 * будет установлен в качестве аргумента типа типа
 * представляющего пропсы компонента [4]
 *
 *
 * [5] определение двух интерфейсов представляющих
 * два различных типа данных.
 *
 * [6] создание экземпляра универсального компонента
 * и установление в качестве пропсов объекты соответствующие [8]
 * и нет [9] требованиям установленными аргументами типа [7].
 */
```

Нет ничего особенного и в определении функционального компонента как *Function Declaration*.

ts

```ts
/**[0] */
interface Props<T> {
    data: T; /**[1] */
}

     /**[2][3]              [4] */
function A <T>(props: Props<T>) {
    return <div></div>;
}

/**
 * [0] определение обобщенного типа чей
 * единственный параметр предназначен для
 * указания в аннотации типа поля data [1].
 *
 * [2] универсальный функциональный компонент
 * определенный как Function Declaration [2] чей
 * единственный параметр типа [3] будет установлен
 * в качестве аргумента типа типа представляющего
 * пропсы компонента [4].
 *
 */
```

Но относительно функциональных компонентов определенных как *Function Expression* не обошлось без курьезов. Дело в том, что в большинстве случаев лучшим способом описания сигнатуры функционального компонента является использование обобщенного типа `FC<P>`. Это делает невозможным передачу параметра типа функции в качестве аргумента типа типу представляющему пропсы, поскольку они находятся по разные стороны от оператора присваивания.

ts

```ts
interface Props<T> {}

const A: FC<Props< /**[0] */ >> = function < /**[1] */ > (props) {
    return <div></div>;
}

/**
 * [0] как получить тут, то...
 * [1] ..., что объявляется здесь?
 */
```

Единственный возможный вариант создания обобщенного функционального компонента определенного как *Function Expression* заключается в отказе от аннотирования идентификатора в пользу типизирования сигнатуры непосредственно компонента.

ts

```ts
interface Props<T> {
    data: T;
}


  /**[0]            [1]          [2] */
const A =  function <T>(props: Props<T>) {
    return <div></div>;
};


<A<number> data={0}/>; // Ok
<A<number> data={''}/>; // Error

/**
 * Чтобы функциональный компонент стал
 * универсальным определение принадлежности
 * идентификатора функционального выражения [0]
 * необходимо поручить выводу типов который
 * сделает это на основе типов явно указанных
 * в сигнатуре функции [1] [2] выступающей в качестве
 * значения.
 */
```

Кроме этого, неприятный момент связан со стрелочными универсальными функциями (*arrow function*) при определении их в файлах имеющих расширение `.tsx`. Дело в том, что невозможно определить универсальную функцию если она содержит только один параметр типа который не расширяет другой тип.

ts

```ts
/**[0] */
const f = <T>(p: T) => {}; /**[1] Error */

[].forEach(/**[2] */<T>() => { }) /**[3] Error */

/**
 * Не имеет значения присвоена универсальная
 * стрелочная функция [0] [2] переменной [1] или определена
 * в месте установления аргумента [3] компилятор
 * никогда не позволит скомпилировать такой код, если
 * он расположен в файлах с расширением .tsx
 */
```

Другими словами, что бы при определении универсальной стрелочной функции в файле с расширением `.tsx` не возникало ошибки её единственный параметр типа должен расширять другой тип...

ts

```
/**[0] */
const f0 = <T extends {}>(p: T) => {}; // Ok

             /**[0] */
[].forEach(<T extends {}>() => { }); // Ok


/**
 * Если единственный параметр типа
 * расширяет другой тип [0], то ошибка
 * не возникает.
 */
```

...либо параметров типа должно быть несколько.

ts

```
/**[0] */
const f0 = <Тб, U>(p: T) => {}; // Ok

          /**[0] */
[].forEach(<T, U>() => { }); // Ok


/**
 *[0] ошибки также не возникает
 если универсальная функция определяет
 несколько параметров типа.
 */
```

Для закрепления информации данной главы выполним небольшой пример. Представьте задачу требующую написание компонента испускающего событие, объект которого содержит свойство `data` хранящего значение переданное вместе с пропсами. Без механизма универсальных компонентов, свойство `data` , как в пропсах, так и объекте событий, будет представлено либо одним конкретным типом, либо множеством типов составляющих тип объединение.

В первом случае, для каждого типа представляющего данные, потребуется определять новый компонент.

ts

```
interface DataEvent<T> {
    data: T;
}


/**[0] */
interface  CardAProps {
    data: number; /**[1] */
```

```
                                /**[1] */
    handler: (event: DataEvent<number>) => void;
}

    /**[2] */
const CardA = ({data, handler}: CardAProps) => {
    return (
        <div onClick={() => handler({data})}>Card Info</div>
    );
}

const handlerA = (event: DataEvent<number>) => {}

<CardA data={0} handler={handlerA} />

/** ============== */

/**[3] */
interface  CardBProps {
    data: string; /**[4] */
                                /**[4] */
    handler: (event: DataEvent<string>) => void;
}

    /**[5] */
const CardB = ({data, handler}: CardBProps) => {
    return (
        <div onClick={() => handler({data})}>Card Info</div>
    );
}

const handlerB = (event: DataEvent<string>) => {}

<CardB data={``} handler={handlerB} />

/**
 * [2] [5] определение идентичных по логике компонентов
 * нужда в которых появляется исключительно из-за необходимости
 * в указании разных типов [1][4] в описании интерфейсов представляющих
 * их пропсы [0][3]
 */
```

Во втором, для сужения множества типов, придется производить утомительные проверки.

ts

```
interface DataEvent<T> {
    data: T;
}
```

```
interface  CardProps {
    data: number | string; /**[0] */
                                  /**[0] */
    handler: (event: DataEvent<number | string>) => void;
}

const Card = ({data, handler}: CardProps) => {
    return (
        <div onClick={() => handler({data})}>Card Info</div>
    );
}


const handler = (event: DataEvent<number | string>) => {
    // утомительные проверки

    if(typeof event.data === `number`){
        // в этом блоке кода обращаемся как с number
    }else if(typeof event.data === `string`){
        // в этом блоке кода обращаемся как с string
    }
}

<Card data={0} handler={handler} />

/**
 * [0] указание типа как объединение number | string
 * избавило от необходимости определения множества компонентов,
 * но не избавила от утомительных и излишних проверок при работе
 * с данными с слушателе событий.
 */
```

Избежать повторяющегося или излишнего кода можно путем определения компонентов универсальными.

ts

```
interface DataEvent<T> {
    data: T;
}


/**[0] */
interface  CardProps<T> {
    data: T; /**[1] */
                          /**[1] */
    handler: (event: DataEvent<T>) => void;
}

    /**[2]            [3]                            [4] */
const Card = function <T>({data, handler}: CardProps<T>) {
    return (
```

```
        <div onClick={() => handler({data})}>Card Info</div>
    );
}

const handlerWithNumberData = (event: DataEvent<number>) => {}
const handlerWithStringData = (event: DataEvent<string>) => {}

<Card<number> data={0} handler={handlerWithNumberData} />;
<Card<string> data={``} handler={handlerWithStringData} />;

/**
 * [2] определение универсального функционального компонента
 * параметр типа которого [3] будет установлен типу представляющего
 * пропсы [0] в качестве аргумента типа [4], что сделает его описание
[1]
 * универсальным.
 */
```

В итоге кода становиться меньше, что делает его проще для чтения. Кроме того, код написанный таким образом более соответствует лучшим канонам обусловленных типизацией.

Глава 54

Типизированные хуки

React api насчитывает десять предопределенных хуков большинство которых являются универсальными. Каждый из них будет рассмотрен по отдельности. Кроме того, данная глава будет посвящена определению пользовательских хуков с учетом последних возможностей *TypeScript*.

[54.00] Предопределенные хуки - useState<T>()

Рассмотрение данной темы стоит начать с самого часто применяемого универсального хука `useState<T>(initialState): [state, dispatch]` параметр типа которого представляет определяемое им состояние. В случаях когда при вызове универсальной функции `useState<T>(initialState)` аргумент типа не устанавливается, тип значения будет выведен на основе аргумента функции обозначаемого `initialState`. Это в свою очередь означает, что при отсутствии инициализационного значения (вызов функции `useState` без значения) или его временном замещении значением принадлежащим к другому типу (например объектный тип замещается значением `null`), или частичном значении (объектный тип определяющий лишь часть своих членов), изменить его в будущем с помощью функции обозначаемой `dispatch` будет невозможно, поскольку она при определении также ассоциируется с выведенным на основе `initialState` типом.

ts

```ts
const A: FC = () => {
                                /**[0][1] */
    let [state, dispatch] = useState();
```

```
    dispatch(0); // Error -> type 0 is not type undefined

    /**
     * Поскольку отсутствие initialState [1] не было
     * компенсировано аргументом типа [0] невозможно
     * установит новое значение с помощью функции dispatch
     * если оно принадлежит к типу отличному от выведенного
     * в момент определения.
     */


    return null;
}

const B: FC = () => {
                                /**[0][1] */
    let [state, dispatch] = useState(null);
    dispatch({ a: 0 }); // Error -> type {a: number} is not type null

    /**
    * Поскольку initialState [1] принадлежит к типу null
    * и отсутствует уточнение типа состояния при помощи
    * аргумента типа [0], то будет невозможно установить новое
    * значение с помощью функции dispatch если оно принадлежит
    * к типу отличному от выведенного в момент определения.
    */


    return null;
}
const C: FC = () => {
                                /**[0] [1] */
    let [state, dispatch] = useState({a: 0});
    dispatch({ a: 0, b: `` }); // Error -> type {a: number, b: string}
is not type {a: number}

    /**
    * Поскольку initialState [1] представляет из себя лишь
    * часть от предполагаемого типа и при этом отсутствует
    * уточнение типа состояния при помощи
    * аргумента типа [0], то будет невозможно установить новое
    * значение с помощью функции dispatch если оно принадлежит
    * к типу отличному от выведенного в момент определения.
    */


    return null;
}
```

В подобных случаях необходимо уточнить тип к которому принадлежит состояние при помощи аргумента типа. Единственное нужно помнить, что несмотря на уточнение типа,

при отсутствии `initialState`, состояние будет принадлежать к объединению `T | undefined`.

ts

```ts
const A: FC = () => {
    let [state, dispatch] = useState<number>();
    dispatch(0); // Ok
    state; // number | undefined


    return null;
}

const B: FC = () => {
                                                /**[*] */
    let [state, dispatch] = useState<{a: number} | null>(null);
    dispatch({ a: 0 }); // Ok
    state; // {a: number} | null


    return null;
}
const C: FC = () => {
                                               /**[*] */
    let [state, dispatch] = useState<{a: number; b?: string;}>({a: 0});
    dispatch({ a: 0, b: `` }); // Ok
    state; // {a: number; b?: string | undefined}


    return null;
}

/**
 * [*] конкретизация типа
 */
```

Все описанные случаи так или иначе предполагают дополнительные проверки на существование значения, которые на практике отягощают код. Поэтому при отсутствии конкретного состояния выступающего в роли аргумента универсальной функции, всегда лучше устанавливать значение по умолчанию в полной мере соответствующего типу, нежели допускать его отсутствие, замещение или частичную установку.

При условии, что `initialState` представлен значением в полной мере соответствующим требуемому типу, необходимость в уточнении с помощью аргумента типа пропадает, поскольку выводу типов не составит особого труда справиться самостоятельно. Простыми словами, если аргумент типа не устанавливается, его тип выводится на основе типа к которому принадлежит аргумент вызываемой функции. В случае, когда состояние в полной мере устанавливается в качестве единственного аргумента хука `useState<T>()` необходимости в уточнении типа, при помощи аргумента функционального типа, не требуется. Если в качестве значения выступает

примитив или объект, все члены которого инициализированны, вывод типов будет только рад взять работу по установлению типа на себя.

ts

```ts
const A: FC = () => {
    let [state, dispatch] = useState(0);
    dispatch(0); // Ok
    state; // number


    return null;
}

const B: FC = () => {
    let [state, dispatch] = useState({a: 0});
    dispatch({ a: 0 }); // Ok
    state; // {a: number}


    return null;
}
const C: FC = () => {
    let [state, dispatch] = useState({a: 0, b: ``});
    dispatch({ a: 0, b: `` }); // Ok
    state; // {a: number; b: string;}


    return null;
}
```

[54.01] Предопределенные хуки - useEffect() и useLayoutEffect()

Следующими на очереди расположились сразу два идентичных с точки зрения типизации хука `useEffect(effect, deps?): void` и `useLayoutEffect(effect, deps?): void`, ожидающие в качестве первого параметра функцию, а в качестве второго необязательного параметра, массив, изменение элементов которого приводит к повторному вызову первого аргумента. Поскольку у данных хуков отсутствует возвращаемое значение, сложно представить сценарий в котором возникает ошибка связанная с передачей аргументов. Поэтому подробное рассмотрение и пояснение будет опущено.

ts

```
const A: FC = () => {
    useEffect(() => {
        return () => {}
    }, []);

    useLayoutEffect(() => {
        return () => {}
    }, []);


    return null;
}
```

[54.02] Предопределенные хуки - useContext<T>()

Следующий претендент на рассмотрение предназначен для работы с контекстом и является универсальной функцией `useContext<T>(context)` принимающей в качестве аргумента объект контекста, который при необходимости можно уточнить с помощью аргумента типа. Поскольку вывод типов в состоянии самостоятельно вывести тип опираясь на обязательный параметр `context`, то уточнение типа с помощью аргумента типа не требуется.

ts

```
import React, {createContext, useContext, FC} from "react";

const StringContext = createContext(`Is Context Value!`);

const A: FC = () => {
    let context = useContext(StringContext // let context: string


    return null;
}
```

Уточнение с помощью аргумента типа может потребоваться только при необходимости приведения более общего типа к более конкретному. Но в реальности универсальная функция этого не позволяет сделать.

ts

```
import React, {createContext, useContext, FC} from "react";
```

```ts
interface T0 {
    a: number;
}
interface T1 {
    b: string;
}


       /**[0] */
interface T2 extends T0, T1 {}

/**
 * [0] более общий тип
 */


let contextDefaultValue: T2 = {
    a: 0,
    b: ``
};

const StringContext = createContext(contextDefaultValue); // const
StringContext: React.Context<T2>

const A: FC = () => {
    let c0 = useContext(StringContext); // let c0: T2
    let c1 = useContext<T0>(StringContext); // Error -> [1]

    /**
     * [1] при попытке приведения более общего типа
     * T2 к более конкретному типу T0 возникает ошибка ->
     * Argument of type 'Context<T2>' is not assignable
     * to parameter of type 'Context<T0>'.
     */

    return null;
}
```

При возникновении потребности в подобном приведении конкретизировать необходимо идентификатор ассоциированный со значением, то есть переменную.

ts

```ts
const A: FC = () => {
    let c2: T0 = useContext(StringContext); // Ok -> let c2: T0

    return null;
}
```

[54.03] Предопределенные хуки - useReducer<R>()

Следующий в списке предопределенных хуков расположился `useReducer<R>(reducer, initialState, stateInit):[state, dispatch]` представленный универсальной функцией имеющей множество перегрузок. Чтобы познакомится с каждым из параметров данной функции, для начала, нам потребуется объявить два типа описывающих состояние (`state`) и инициализационное состояние (`initialState`), которое специально будет отличаться от обычного, что бы преобразовать его при помощи функции обозначенной как `stateInit` .

ts

```ts
/**[0] */
interface InitialState {
    name: string;
    age: number;
    gender: `male` | `female` | `notSpecified`;
}
/**[1] */
interface State {
    name: string;
    gender: `male` | `female` | `notSpecified`;
}

/**
 * Объявление интерфейсов описывающих
 * состояние [1] и инициализационное состояние [0].
 *
 */
```

Затем определим функцию `reducer` для описания сигнатуры которой воспользуемся импортированным их пространства имен *React* обобщенным типом `Reducer<S, A>` ожидающего в качестве первого аргумента типа тип описывающий `state` , а в качестве второго тип описывающий действие `Action` . В нашем примере второй аргумент типа `Reducer<S, A>` будет представлен псевдонимом для двух конкретных типов действий, перед объявлением которых стоит обратить внимание на один относящийся к ним тонкий момент. Тонкость заключается в том, что функция-редюсер, в качестве второго параметра, может, и в большинстве случаев будет принимать объекты действий принадлежащих к разным типам. Для их конкретизации выводу типов потребуется помощь в виде дискриминантных полей.

ts

```
import React, {useReducer, Reducer, FC} from "react";

interface InitialState {
    name: string;
    gender: number;
    gender: `male` | `female` | `notSpecified`;
}
interface State {
    name: string;
    gender: `male` | `female` | `notSpecified`;
}


/**[0] */
enum ActionType {
    Name = `name`,
    Gender = `gender`,
}

/**
 * [0] определение перечисления содержащего
 * константы необходимые для конкретизации типа
 * Action.
 */


            /**[1] */
interface NameAction {
    type: ActionType.Name; /**[2] */
    name: string; /**[3] */
}
            /**[1] */
interface GenderAction {
    type: ActionType.Gender; /**[2] */
    gender: `male` | `female` | `notSpecified`; /**[3] */
}

/**
 * [1] объявление более конкретных типов действий
 * объявляющий поля name и gender [3] и дискриминантное
 * поле type [2] в качестве типа которого указан элемент
 * перечисления.
 */


/**[4] */
type Actions = NameAction | GenderAction;

/**
 * [4] объявление псевдонима ссылающегося на
 * тип объединение представленный типами Action.
 */
```

```
     /**[5]      [6]    [7]     [8]            [9]     [10] */
const reducer: Reducer<State, Actions> = (state, action) => {
                /**[11] */
    if(action.type === ActionType.Name){
                                /**[12] */
        return {...state, name: action.name};
                    /**[11] */
    }else if(action.type === ActionType.Gender){
                                /**[12] */
        return {...state, gender: action.gender};
    }


    return state;
}

/**
 * [5] определение функции редюсера сигнатура
 * которой уточняется при помощи импортированного
 * из пространства имен React обобщенного типа Reducer<S, A> [6]
 * в качестве первого параметра который получает тип представляющей
 * State [7], а в качестве второго тип объединение Actions [8].
 * При таком сценарии сигнатура функции в явном указании типов не
нуждается [9] [10]
 *
 * Блок кода, вхождение в который возможно в результате положительного
 * результата выполнения условия на принадлежность дискриминантного
поля type
 * элементу перечисления [11], подразумевает, что объект action
принадлежит
 * к соответствующему типу, что позволяет обращаться к присущим только
ему членам [12]
 *
 */
```

Как было сказано ранее, сигнатура редюсера не нуждается в аннотации если его тип конкретизирован с помощью `Reducer<S, A>`. За этим скрывается один коварный момент. Представьте ситуацию при которой некоторое время приложение работало с состоянием определяющим `name`, а затем было принято решение изменить его на `fullName`.

ts

```
// было

interface State {
    name: string;
    age: number;
}
```

```
interface NameAction {
    type: `name`;
    name: string;
}

const reducer: Reducer<State, NameAction> = (state, action) => {
    if(action.type === `name`){
        return {...state, name: action.name};
    }

    return state;
}
```

После того как тип описывающий состояние и действие претерпят изменения, неминуемо возникнет ошибка указывающая, что в действии больше нет поля `name`. Если впопыхах изменить лишь старый идентификатор `name` на новый `fullName`, то можно не заметить как значение ассоциированное с новым идентификатором определенным в объекте действия присваивается старому идентификатору определяемому в объекте нового состояния. К ошибке это не приведет, поскольку механизм распространения старого состояния `...state` наделяет новое всеми признаками необходимыми для совместимости с типом `State`. Старый идентификатор `name` в новом состоянии делает его принадлежащим к более общему типу который совместим с объектом `State`. Это неминуемо приводит к трудновыявляемому багу, поскольку значение поля `fullName` всегда будет старым.

ts

```
interface State {
    fullName: string; /**[0] */
    age: number;
}

interface NameAction {
    type: `name`;
    fullName: string; /**[0] */
}

const reducer: Reducer<State, NameAction> = (state, action) => {
    if(action.type === `name`){
                        /**[1]        [2] */
        return {...state, name: action.fullName};
    }

    return state;
}

/**
 * При изменении State и NameAction [0] ошибка
 * укажет на отсутствие поля name  в объекте
 * принадлежащего к типу NameAction [2]. Интуитивное
 * исправление лишь этой ошибки приведет к трудновыявляемому
 * багу, поскольку новое поле действия присваивается старому
```

```
идентификатору [1].
 * Нужно быть внимательным, поскольку ошибки не возникнет. Причина
заключается в том
 *, что распространение старой версии ...state наделяет новый объект
всеми необходимыми
 * характеристиками, что бы быть совместимым с типом State. Старое поле
делает из нового объекта
 * значение принадлежащие к более общему типу который совместим с типом
State. Поэтому
 * ошибки связанной с неверным возвращающимся значением не возникает.
Тем не менее редюсер
 * никогда не изменит значение на новое. Оно всегда будет браться из
предыдущего объекта состояния.
 */
```

Возвращаясь к основному примеру осталось определить компонент в котором и будет происходить определение элементов *Redux*. Первым делом в теле компонента определим инициализационное состояние которое с помощью функции обозначенной ранее как `stateInit` будет преобразовано в значение соответствующее типу необходимого редюсеру. Стоит заметить, что поскольку инициализационное значение в полной мере соответствует типу `InitialState` аннотация типа является излишней. При определении с помощью универсальной функции `useReducer()` элементов редакса стоит сделать акцент на отсутствии необходимости в указании аргументов типа, поскольку вывод типов будет опираться на аннотации типов параметров данного хука.

Осталось последовательно изменить состояние с помощью функции `dispatch` вызов которой с объектами в точносте соответствующих типам `*Action` не вызывает никаких нареканий. Вызов с аргументом не надлежащего типа приводит к возникновению ошибки, что в свою очередь подтверждает надежность описанной логики сопряженной с хуком `useReducer()` .

ts

```ts
/**[13]                [14] */
const initState = (initialState: InitialState) => {
    let {gender, ...reducerState} = initialState;

    return reducerState; /**[15] */
};

const A: FC = () => {
        /** [16]        [17] */
    let initialState: InitialState = {
        name: `noname`,
        age: NaN,
        gender:  `notSpecified`
    };
                        /**[18] */
    let [state, dispatch] = useReducer(reducer, initialState,
initState);

    /**[19] */
```

```
    dispatch({
        type: ActionType.Name,
        name: `superuser`
    });

    /**[19] */
    dispatch({
        type: ActionType.Gender,
        gender: `male`
    });

    /**[20] */
    dispatch({
        type: ActionType.Gender,
        gender: `male`,
        age: 100
    }); // Error -> Object literal may only specify known properties,
and 'age' does not exist in type 'GenderAction'.


    return null;
}

/**
 * [14] определение инициализационного состояния которое будет
 * передано в качестве единственного аргумента [14] функции initState
[13]
 * предназначенной для его трансформации в состояние пригодное редюсеру
[15].
 * Данная функция вместе с редюсером и инициализационным состоянием
передается
 * в универсальную функцию useReducer в качестве аргументов. Стоит
заметить, что
 * универсальная функция useReducer определяющая элементы redux [18]
 * в указании аргументов типа не нуждается так как вывод типов
отталкивается от
 * типов указанных в аннотациях элементов ожидаемых в качестве
параметров.
 *
 * [19] успешное изменение состояния.
 * [20] попытка изменить состояние с помощью объекта тип которого не
совместим
 * с типом State приводит к ошибке.
 */
```

Не будет лишним пролить свет на подход объявления типов действий в основе которых лежит базовый тип. Подобное часто требуется для так называемых `DataAction`, действий которые помимо поля `type` определяют ещё и поле `data` с типом `T`.

Для этого потребуется объявить два обобщенных типа. Первым объявим обобщенный тип `Action<T>`, единственный параметр типа которого будет указан в аннотации типа дискриминантного поля `type`. Вторым тип `DataAction<T, D>`, первый параметр

типа которого будет передан при расширении в качестве аргумента типу `Action<T>`, а второй параметр типа будет указан в аннотации типа единственного поля `data`. Такой подход часто применяется на практике и значительно упрощает объявление типов представляющих действия предназначенных исключительно для транспортировки данных.

ts

```
/**[0] */
interface Action<T> {
    type: T;
}
/**[1] */
interface DataAction<T, D> extends Action<T> {
    readonly data: D;
}

/**
 * Объявление обобщенных типов действий первый из которых
 * описывает обычное действие с единственным полем
 * type: T [0], а другой расширяет первый и определяет
 * поле data: D.
 */



enum ActionType {
    Name = `name`,
    Gender = `gender`,
}


interface NameData {
    name: string;
}
interface GenderData {
    gender: `male` | `female` | `notSpecified`;
}


         /**[2]                [3]           [4]          [5] */
interface NameAction extends DataAction<ActionType.Name, NameData> {
}
         /**[2]                [3]           [4]               [5] */
interface GenderAction extends DataAction<ActionType.Gender, GenderData>
{
}

/**
 * [2] определение конкретных действий в основе
 * которых лежит базовый обобщенный тип DataAction<T, D> [3]
 * которому при расширении в качестве первого аргумента типа
 * устанавливается тип дискриминантного поля [4], а в качестве
```

```
 * второго тип представляющий данные [5].
 */

type Actions = NameAction | GenderAction;
```

Осталось рассмотреть только случай предполагающий указания аргументов типа универсальной функции острая необходимость в которых возникнет при определении аргументов в момент вызова. В таком случае в качестве первого аргумента типа универсальная функция ожидает тип описывающий функцию редюсер, вторым инициализационное состояние. Кроме того, прибегнуть к помощи аргументов типа может потребоваться и при других сценариях рассматриваемых на всем протяжении темы посвященной хукам. Также не стоит забывать, что универсальная функция `useReducer` имеет множество перегрузок, что на практике допускает сценарии отличные от данного примера.

ts

```
import React, {useReducer, Reducer, FC, useRef} from "react";

interface InitialState {
    name: string;
    age: number;
    gender: `male` | `female` | `notSpecified`;
}
interface State {
    name: string;
    gender: `male` | `female` | `notSpecified`;
}


enum ActionType {
    Name = `name`,
    Gender = `gender`,
}

enum Genders {
    Male = `male`,
    Female = `female`,
    NotSpecified = `notSpecified`
}


interface NameAction {
    type: ActionType.Name;
    name: string;
}
interface GenderAction {
    type: ActionType.Gender;
    gender: Genders;
}
```

```
type Actions = NameAction | GenderAction;

const App: FC = () => {
                                                /**[0]                    [1]
*/
    let [state, dispatch] = useReducer<Reducer<State, Actions>,
InitialState>(
        (state, action) => state, /**[2] */
        {name: ``, age: 0, gender: Genders.NotSpecified}, /**[3] */
        (initialState) => ({name: initialState.name, gender:
initialState.gender})
    );

    return null;
}

/**
 * Указание аргументов типа универсальной функции
 * требуется тогда, когда её аргументы определяются
 * при вызове. В таком случае первый аргумент типа
 * будет представлять описание [0] функции редюсера [02],
 * второй инициализационного состояния [1] [3]. Также возможны
 * и другие сценарии требующий указания аргументов типа которые
 * были затронуты ранее в теме. Кроме того, универсальная функция
 * имеет множество перегрузок допускающих отличие от данного примера.
 */
```

[54.04] Предопределенные хуки - useCallback<T>()

Следующий на очереди универсальный хук `useCallback<T>(callback: T, deps): T` рассмотрение которого можно ограничить иллюстрацией его применения, поскольку с ним не связанно ничего, что к данному моменту могло бы вызвать хоть какой-то вопрос. Другими словами, как и в остальных рассмотренных ранее случаях прибегать к помощи аргументов типа следует только тогда, когда сигнатура функции обозначенной как `callback` частично или вовсе не имеет указания типов.

ts

```
import React, {useCallback, FC} from "react";


const A: FC = () => {
    /**[0] */
    const greeter = useCallback((userName: string) => {
```

```
        return `Hello ${userName}!`;
    }, []);

    /**
     * [0] вывод типов ->
     *
     * const greeter: (userName: string) => string
     */


    return null;
}



type Decorate = (username: string) => string;

const B: FC = () => {
    /**[0] */
    const greeter = useCallback<Decorate>( userName => {
        return `Hello ${userName}!`;
    }, []);

    /**
     * [0] аргумент типа ->
     *
     * const greeter: Decorate
     */


    return null;
}
```

[54.05] Предопределенные хуки - useRef<T>()

Следующий хук `useRef<T>(initialValue)` является универсальной функцией предназначенной для создания объекта рефы. Поскольку объект рефы возвращаемый из данного хука может служить не только для хранения ссылок на *React элементы* и *React компоненты*, но и на любые другие значения, которые к тому же могут выступать в качестве единственного аргумента, данная функция имеет множество перегрузок. И кроме этого на текущий момент с ней связан один нюанс. Но обо всем по порядку.

В тех случаях когда объект рефы выполняет роль хранилища для не относящихся к *React* значений, в его определении нет ничего необычного. Простыми словами, если инициализационное значение устанавливается частично или не устанавливается вовсе,

то его тип необходимо конкретизировать с помощью аргумента типа. В противном случае переложить эту работу на вывод типов.

ts

```ts
import React, {useRef, FC} from "react";

interface Data {
    a: number;
    b: string;
}

const A: FC = () => {
                          /**[0]   [1]      [2] */
    let dataRef = useRef<Partial<Data>>({a: 0}); // let dataRef:
React.MutableRefObject<Partial<Data>>
    let a = dataRef.current.a; // let a: number | undefined

    return null;
}

/**
 * Поскольку значение [1] установлено частично [2]
 * возникла необходимость прибегнуть к помощи типа
 * Partial<T> [0].
 */

const B: FC = () => {
                               /**[0]        [1] */
    let dataRef = useRef<Data | undefined>(); // let dataRef:
React.MutableRefObject<Data | undefined>
    let a = dataRef.current?.a; // let a: number | undefined

    return null;
}

/**
 * Поскольку значение [1] вовсе не было установлено
 * аргумент типа указан как объединение включающего
 * тип undefined [0].
 */

const C: FC = () => {
           /**[0]      [1]   [2] */
    let dataRef = useRef({a: 0, b: ``}); // let dataRef:
React.MutableRefObject<{a: number;b: string;}>
    let a = dataRef.current.a; // let a: number

    return null;
}

/**
 * Поскольку значение [2] в полной мере соответствует
```

```
 * предполагаемому типу необходимости в конкретизации
 * типа отпадает [0] [1].
 */
```

Также стоит обратить внимание, что идентификатор которому в момент определения присваивается результат вызова функции `useRef()` в явной аннотации не нуждается.

ts

```ts
const C: FC = () => {
        /**[0] */
    let dataRef = useRef({a: 0, b: ``});

    return null;
}

/**
 * [0] ссылка на возвращаемый хуком useRef объект
 * в аннотации типа не нуждается. Указание аннотаций
 * типа в подобных случаях лишь отнимают время и
 * затрудняют чтение кода.
 *
 * let dataRef: React.MutableRefObject<{a: number;b: string;}>
 */
```

Осталось оговорить упомянутый в самом начале нюанс который связан с объектом рефы устанавливаемого *React элементу*. Дело в том, что декларация *React элементов*, в частности поля `ref`, аннотирована устаревшим типом, который не совместим с типом значения получаемого в результате вызова хука `useRef()`. К тому же уточнение типа в аннотации идентификатора или с помощью аргументов типа универсальной функции с данной проблемой справится не помогут. Единственное решение явное приведение к типу `RefObject<T>` возвращаемое хуком значение с помощью оператора `as`.

ts

```ts
import React, {useRef, FC, RefObject} from "react";

const A: FC = () => {
    // let formRef: RefObject<HTMLFormElement> | null = useRef(); //
Error
    // let formRef = useRef<RefObject<HTMLFormElement> | null>(); //
Error
    let formRef = useRef() as RefObject<HTMLFormElement> | null; // Ok


    return <form ref={formRef}></form>
}
```

Более подробно с данным хуком можно познакомиться в главе посвященной рефам.

[54.06] Предопределенные хуки - useImperativeHandle<T, R>()

Следующий на очереди хук `useImperativeHandle<T, R>(ref: Ref<T>, apiFactory() => R): void` предназначенный для присваивания объекта выступающего в роли открытой части *api* функционального компонента. В тех случаях когда открытая часть *api* отличается от закрытой, универсальная функция нуждается в уточнении при помощи аргументов типа. В остальных случах всегда рекомендуется поручить эту работу выводу типов.

ts

```ts
import React, {ForwardRefRenderFunction, RefObject, useRef,
useImperativeHandle, FC, forwardRef} from "react";

interface FormProps {

}

/**[0] */
interface PublicFormApi {
    a: () => void;
}
/**[1] */
interface PrivateFormApi extends PublicFormApi {
    b: () => void;
}

/**
 * [0] объявление типа представляющего открытую часть
 * api компонента, которую будет расширять объявленный
 * позже тип описывающий его закрытую часть [1].
 */


 /**
  *
  * [!] не обращать внимание на код
  * помеченный восклицательным знаком,
  * поскольку данные участи кода будут
  * подробно рассмотрены в свое время.
  */

                    /**[!] */
```

```
const Form: ForwardRefRenderFunction<PublicFormApi, FormProps> = (props,
ref) => {
                          /**[2]              [3] */
    useImperativeHandle<PublicFormApi, PrivateFormApi>(ref, () => ({
        a: () => {},
        b: () => {},
    }), []);

    /**
     * С помощью аргументов типа указываем тип
     * открытой [2] и закрытой [3] части api компонента.
     */


    return null;
}

        /**[!]            [!] */
const FormWithRef = forwardRef(Form);


const App: FC = () => {
                        /**[!] */
    let formRef = useRef() as RefObject<PublicFormApi>;

    formRef.current?.a(); // Ok
    // formRef.current?.b(); // Error

                     /**[6] */
    return <FormWithRef ref={formRef} />;
}
```

[54.07] Предопределенные хуки - useMemo<T>()

Следующим на очереди хук `useMemo<T>(factory): T` является универсальной функцией ожидающей в качестве первого параметра фабричную функцию вычисляющую возвращаемое хуком значение тип которого, при необходимости, можно указать с помощью аргумента типа. Второй обязательный параметр принадлежит к типу массива, при наличии и изменении элементов которого происходит повторный вызов фабричной функции. Как всегда, стоит упомянуть, что явное указание аргумента типа универсальной функции необходимо в очень редких случаях, каждый из которых был рассмотрен на протяжении всей темы посвященной хукам. Кроме того, переменная которой присваивается возвращаемое из хука значение, в аннотации типа и вовсе не нуждается.

ts

```ts
import React, {useMemo, FC} from "react";

const A: FC = () => {
            /**[0]           [1] */
    let memoizedValue = useMemo( () => 2 + 2, [] ); // let
memoizedValue: number

    return null;
}

/**
 * Нет необходимости указывать аннотацию
 * типа переменной [0] и передавать аргументы
 * типа [1] универсальной функции поскольку
 * вывод типов самостоятельно справится с этой работой.
 */
```

[54.08] Предопределенные хуки - useDebugValue<T>()

Следующий и последний на очереди хук `useDebugValue<T>(data: T, format(data: T) => any): void` предназначенный для вывода логов в *devtools* является универсальной функцией первый параметр которой ожидает выводимое в консоль значение, чьё форматирование может быть осуществлено при помощи функции выступающей в роли второго необязательного параметра. Вся информация имеющаяся к этому моменту и касающаяся явного указания типов в полной мере справедлива и для текущего хука.

ts

```ts
import React, {useDebugValue, FC} from "react";

const A: FC = () => {
                    /**[0]    [1]           [2] */
   useDebugValue(new Date(), date => date.getMilliseconds() );

   return null;
}

/**
* Хук принимает значение тпа Date [0]
* и затем передает его в функцию форматирования [1]
```

```
* из которой возвращается значение принадлежащее к
* типу number [2].
*/
```

[54.09] Пользовательский хук

Помимо предопределенных хуков рассмотренных ранее, *React* также позволяет определять пользовательские хуки, которым будет посвящен остаток текущей главы.

Чтобы стразу перейти к делу представьте, что перед разработчиком встала задача реализовать эффект печатающегося текста с возможностью его запускать, останавливать и ставить на паузу при необходимости. Для этого потребуется реализовать пользовательский хук `usePrintText(text, interval)`, который на вход будет принимать исходный текс и значение скорости печати. Поскольку печатающийся текст будет реализован за счет постепенного изменения состояния определенного внутри хука и выдаваемого наружу, начать стоит именно с описания его типа.

ts

```ts
/**описание объекта состояния */
interface TextPrintInfo {
    isDone: boolean;
    status: TextPrintStatus;
    currentText: string;
    sourceText: string;
}
```

Кроме состояния, хук должен предоставлять функции выступающие в роли контролов предназначенных для управления анимацией печатанья. Поэтому вторым шагом потребуется описать тип представляющих контролы, что совершенно не составит никакого труда, так как старт\пауза\стоп являются обычными функциями которые ничего не возвращают.

ts

```ts
/**описание типа контролов старт\пауза\стоп */
type ControlCallback = () => void;
```

Осталось описать тип представляющий сам хук или если быть точнее, сигнатуры функции, которой по сути он является. Особое внимание стоит обратить на возвращаемый тип представленный размеченным кортежем, для которого несмотря на

большое количество кода не был создан псевдоним. Такое решение было принято из-за того, что автокомплит *ide* указал бы возвращаемый тип как псевдоним, сделав тем самым подсказку малоинформативной, лишив её информации, доступной благодаря меткам.

ts

```ts
/**[0]                                                    [1] */
type UsePrintText = (sourceText: string, interval?: number) => [
    textInfo: TextPrintInfo,
    start: ControlCallback,
    pause: ControlCallback,
    stop: ControlCallback
];

/**
 * [0] тип представляющий пользовательский хук,
 * тип возвращаемого значения представлен размеченным кортежем [1]
 */
```

Осталось лишь определить сам хук.

ts

```ts
interface TextPrintInfo {
    isDone: boolean;
    status: TextPrintStatus;
    currentText: string;
    sourceText: string;
}

type ControlCallback = () => void;

type UsePrintText = (sourceText: string, interval?: number) => [
    textInfo: TextPrintInfo,
    start: ControlCallback,
    pause: ControlCallback,
    stop: ControlCallback
];



const usePrintText: UsePrintText = (sourceText, interval = 200) => {
    // определение состояния
    let [textInfo, setTextInfo] = useState({
        isDone: false,
        status: TextPrintStatus.Waiting,
        currentText: ``,
        sourceText,
    });
```

```
    // определение контролов
    const start = () => {
    }
    const pause = () => {
    }
    const stop = () => {
    }


    return [textInfo, start, pause, stop];
}
```

Поскольку логика работы хука не имеет никакого отношения к *TypeScript*, её детального разбора не будет. Тем не менее полный код представлен и при желании испытать свои знания, предлагается самостоятельно устно его прокомментировать.

ts

```
import React, { useState, useEffect } from "react";

enum TextPrintStatus {
    Print = `print`,
    Waiting = `waiting`,
    Pause = `pause`,
    Done = `done`
}

interface TextPrintInfo {
    isDone: boolean;
    status: TextPrintStatus;
    currentText: string;
    sourceText: string;
}

type ControlCallback = () => void;
type UsePrintText = (sourceText: string, interval?: number) => [
    textInfo: TextPrintInfo,
    start: ControlCallback,
    pause: ControlCallback,
    stop: ControlCallback
];

const usePrintText: UsePrintText = (sourceText, interval = 200) => {
    let [timeoutId, setTimeoutId] = useState(NaN);
    let [textInfo, setTextInfo] = useState({
        isDone: false,
        status: TextPrintStatus.Waiting,
        currentText: ``,
        sourceText,
    });
```

```
    const isDone = () => textInfo.currentText.length ===
sourceText.length;
    const getNextText = () =>
textInfo.currentText.concat(sourceText.charAt(textInfo.currentText.length)

    useEffect(() => {
        if (textInfo.status === TextPrintStatus.Print && !
textInfo.isDone) {
            print();
        }

    }, [textInfo]);

    useEffect(() => () => cancel(), []);


    const print = () => {
        let timeoutId = setTimeout(() => {
            setTextInfo({
                status: isDone() ? TextPrintStatus.Done :
TextPrintStatus.Print,
                isDone: isDone(),
                currentText: getNextText(),
                sourceText
            });
        }, interval);

        setTimeoutId(timeoutId);
    }

    const cancel = () => {
        if (!Number.isNaN(timeoutId)) {
            clearTimeout(timeoutId);
        }
    }
    const start = () => {
        setTextInfo({
            ...textInfo,
            status: TextPrintStatus.Print,
        });
    }
    const pause = () => {
        cancel();
        setTextInfo({
            ...textInfo,
            status: TextPrintStatus.Pause,
            isDone: false,
        });
        setTimeoutId(NaN);
    }
    const stop = () => {
        cancel();
        setTextInfo({
```

```
            isDone: false,
            status: TextPrintStatus.Waiting,
            currentText: ``,
            sourceText
        });
        setTimeoutId(NaN);
    }


    return [textInfo, start, pause, stop];
}



const App = () => {
    let [{currentText}, start, pause, stop] = usePrintText(`React +
TypeScript = ♥`);


    return (
        <>
            <p>{currentText}</p>
            <button onClick={() => start()}>start</button>
            <button onClick={() => pause()}>pause</button>
            <button onClick={() => stop()}>stop</button>
        </>
    )
}
```

Глава 55

Контекст (Context)

Данная глава посвящена рассмотрению влияния типизации на такой механизм как *контекст*, который хотя и не привносит ничего, что на текущий момент могло бы вызвать удивление, все же имеет один не очевидный момент.

[55.0] Определение контекста

Определение контекста осуществляется при помощи универсальной функции определяющей один обязательный параметр выступающий в качестве инициализационного значения и возвращающей объект контекста `createContext<T>(initialValue: T): Context<T>` . В случаях когда инициализационное значение в полной мере соответствует предполагаемому типу, чьё описание не включает необязательных членов, то аргументы типа можно или даже нужно не указывать. В остальных случаях это становится необходимостью.

ts

```ts
import {createContext } from "react";


/**Аргумента типа не требуется */

interface AContext {
    a: number;
    b: string;
}
```

```
         /**[0]             [1][2] */
export const A = createContext({
    a: 0,
    b: ``
});

/**
 * Поскольку при определении контекста [0]
 * в качестве обязательного аргумента было
 * установлено значение [2] полностью соответствующее
 * предполагаемому типу AContext, аргумент типа
 * универсальной функции можно опустить [1].
 *
 */


 /**Требуется аргумент типа */

interface BContext extends AContext {
    c?: boolean;
}

                                /**[0]     [1] */
export const B = createContext<BContext>({
    a: 0,
    b: ``
});


/**
 * Так как инициализационное значение [1]
 * лишь частично соответствует предполагаемому
 * типу BContext тип объекта контекста необходимо
 * конкретизировать при помощью аргументов типа [0]
 */


/**Требуется аргумент типа */

                                      /**[0]       [1] */
export const C = createContext<BContext | null>(null);

/**
 * По причине отсутствия на момент определения
 * инициализационного значения оно заменено на null [1],
 *, что требует упомянуть при конкретизации типа значения [0].
 */
```

[55.1] Использование контекста

Поскольку функциональные и классовые компоненты подразумевают различные подходы для взаимодействия с одними механизмами реализуемыми *React*, после регистрации контекста в *react дереве* работа с ним зависит от вида компонента нуждающегося в поставляемых им данных.

После определения контекста необходимо зарегистрировать в *react дереве* предоставляемого им `Provider` установив ему необходимые данные.

ts

```
import React, {createContext } from "react";

/**0 */
const Context = createContext({
    status: ``,
    message: ``
});

/**
 * [0] определение контекста.
 */

   /**[1] */
const App = () => (
        /**[2]                                    [3] */
    <Context.Provider value={{status: `init`, message: `React Context!
`}}>

    </Context.Provider>
)

/**
 * [1] определяем компонент представляющий точку входа
 * в приложение и регистрируем в его корне Provider [2]
 * которому при инициализации устанавливаем необходимое значение [3].
 */
```

На следующем шаге определим классовый компонент и добавим его в ветку, корнем которой является определенный на предыдущем шаге `Provider` .

ts

```
const App = () => (
    <Context.Provider value={{status: `init`, message: `React Context!
`}}>
        <ClassComponent />
    </Context.Provider>
)

class ClassComponent extends Component {
    render(){
        return (
        );
    }
}
```

Поскольку компонент является классовым, единственный способ добраться до предоставляемых контекстом данных заключается в создании экземпляра `Consumer`, который в качестве `children` ожидает функцию обозначаемую как `render callback`. Данная функция определяет единственный параметр принадлежащий к типу данных передаваемых с помощью контекста, а возвращаемое ею значение должно принадлежать к любому допустимому типу представляющему элемент *React дерева*.

ts

```
class ClassComponent extends Component {
    render(){
        return (
                  /**[0]        [1]               [2] */
            <Context.Consumer >{data => <span>{data.message}</span>}</
Context.Consumer>
        );
    }
}

/**
 * Поскольку компонент ClassComponent является
 * классовым, единственный вариант получить в нем
 * данные предоставляемые контекстом заключается
 * в создании экземпляра Consumer, который в качестве
 * children ожидает функцию обозначаемую как render callback
 * единственный параметр которой принадлежит к типу данных, а
 * возвращаемое значение должно принадлежать к одному из допустимых
 * типов предполагаемых React.
 */
```

Случаи предполагающие определение `render callback` вне `Consumer` потребуют указания аннотации типа его единственному параметру, тип для которого лучше объявить в месте определения контекста. Если данные инициализации в полной мере соответствуют ожидаемому типу, то его получение проще выполнить с помощью механизма *запроса типа*, чем описывать вручную. В остальных случаях описание потребуется выполнять самостоятельно.

ts

```
/**0 */
let initialValue = {
    status: ``,
    message: ``
};
const Context = createContext(initialValue);

      /**[1]         [2] */
type ContextType = typeof initialValue;

/**
 * [0] определение инициализационного значения,
 * на основе которого при помощи запроса типа [2]
 * будет получен его тип [1].
 */
```

Полученный тип необходимо будет указать в аннотации единственного параметра `render callback` при его определении.

typeScript

```
class ClassComponent extends Component {
    /**[0]                        [1] */
    renderCallback = (data: ContextType) =&gt; (
        &lt;span&gt;{data.message}&lt;/span&gt;
    );


    render(){
        return (
                                    /**[2] */
            &lt;Context.Consumer &gt;{this.renderCallback}&lt;/
Context.Consumer&gt;
        );
    }
}

/**
 * При внешнем [2] определении render callback как поля класса [0]
 * в аннотации тип его единственного параметра указан тип данных [1]
 * предоставляемых контекстом.
 *
 */
```

Если данные предоставляемые контекстом принадлежать к более общему типу, то параметр `render callback` можно конкретизировать.

ts

```
/**[0] */
interface Message {
    message: string;
}

/**[0] */
interface Status {
    status: string;
}

      /**[1]             [2] */
type ContextType = Message & Status;


let initialValue = {
    status: ``,
    message: ``
};
const Context = createContext(initialValue);



/**
 * [0] объявление конкретных типов
 * определяющих тип пересечение [2]
 * на который ссылается прежний псевдоним [1].
 *
 * Поскольку инициализационное значение в полной
 * мере соответствует предполагаемому типу, переменную
 * initialValue и универсальную функцию можно избавить от
 * явной и излишней конкретизации.
 */


 class ClassComponent extends Component {
                             /**[3] */
    renderCallback = (data: Message) => (
        <span>{data.message}</span>
    );


    render(){
        return (
            <Context.Consumer >{this.renderCallback}</Context.Consumer>
        );
    }
}

/**
 * [3] параметр render callback теперь ограничен типом
 * Message.
 */
```

Для получения данных распространяемых контекстом внутри тела функционального компонента, помимо варианта с `Consumer`, который ничем не отличается от рассмотренного в этой теме ранее, предусмотрен более предпочтительный способ предполагающий использование предопределенного хука `useContext<T>(context)`.

Универсальная функция `useContext` ожидает в качестве своего единственного аргумента объект контекста, конкретизировать который с помощью аргумента типа не имеет никакого смысла.

ts

```
const FunctionComponent = () => {
    let {message, status} = useContext(Context);


    return (
        <span>{message}</span>
    );
}

const App = () => (
    <Context.Provider value={{status: `init`, message: `React Context!
`}}>
        <FunctionComponent />
    </Context.Provider>
)
```

При попытке с помощью аргумента типа ограничить более общий тип данных возникнет ошибка, поскольку в действие включаются правила контравариантности параметров функции.

ts

```
const FunctionComponent = () => {
                              /**[0]      [1] */
    let {message} = useContext<Message>(Context); // Error


    return (
        <span>{message}</span>
    );
}

/**
 * При попке ограничить тип с помощью аргумента типа [0]
 * из-за контравариантности параметров функции возникнет ошибка [1].
 */
```

Глава 56

HOC (Higher-Order Components)

Настало время рассмотреть со всех сторон механизм предназначенный для расширения функциональных возможностей компонента с помощью компонента-обертки обозначаемого как *Higher-Order Components* или сокращенно *HOC*.

[56.0] Определение hoc

Раньше, при разработке *React* приложений разработчикам часто приходилось создавать конструкцию, известную в *react* сообществе, как *HOC* (*Higher-Order Components*).

HOC — это функция, которая на входе принимает один компонент, а на выходе возвращает новый с более расширенным функционалом. Другими словами, *hoc* — это функция, ожидающая в качестве параметров компонент (назовем его входным), который оборачивается в другой, объявленный в теле функции, компонент, выступающий в роли возвращаемого из функции значения (назовем его выходным). Слово “оборачивание”, применимое относительно компонентов, означает, что один компонент отрисовывает (рендерит) другой компонент, со всеми вытекающими из этого процесса (проксирования). За счет того, что входной компонент оборачивается в выходной, достигается расширение его и/или общего функционала. Кроме того, это позволяет устанавливать входному компоненту как зависимости, так и данные, полученные из внешних сервисов.

[56.1] Определение hoc на основе функционального компонента

В качестве примера реализуем сценарий при котором пропсы *компонента обертки* определяемого в теле *hoc* разделяются на две категории. Первая необходима исключительно самому компоненту-обертке для генерации новых пропсов, которые в дальнейшем будут объединены с пропсами относящихся ко второй категории и установлены *оборачиваемому компоненту*.

Начать стоит с детального рассмотрения сигнатуры универсальной функции, ожидающей в качестве единственного параметра типа тип `WrappedProps` представляющий пропсы предназначенные исключительно оборачиваемому-компоненту ссылка на который доступна через единственный параметр `WrappedComponent`. `WrappedComponent` может принадлежать, как к функциональному `FC<T>`, так и классовому типу `ComponentClass<T>`, поэтому указываем ему в аннотации обобщенный тип `Component<P>` пропсы которого, помимо типа представленного аргументом типа `WrappedProps`, должны принадлежать ещё и к типу `WrapperForWrappedProps` описывающего значения создаваемые и устанавливаемые компонентом-оберткой.

Стоит упомянуть, что тип `Component<P>` является типом объединением представляющим классовые и функциональные *React* компоненты.

Поскольку в нашем конкретном примере *функция hoc* в качестве компонента-обертки определяет функциональный компонент, тип возвращаемого значения указан соответствующим образом `FC<T>`. Пропсы компонента-обертки должны принадлежать к нескольким типам одновременно поскольку для его работы требуются не только пропсы необходимые исключительно ему (`WrapperProps`), но и пропсы которые он лишь пробрасывает оборачиваемому-компоненту (`WrappedProps`). Поэтому аргумент типа представляющего возвращаемое значение является типом пересечение `WrappedProps & WrapperProps`.

ts

```ts
/**[0] */
import React, {ComponentType} from "react";


/**[1] */
export interface WrapperProps {
    a: number;
    b: string;
```

```
}
/**[2] */
export interface WrapperForWrappedProps {
    c: boolean;
}

/**
 * [0] Импортируем обобщенный тип Component<P> представляющий
 * объединение классового и функционального React компонента.
 * [1] WrapperProps описывает данные необходимые
 * исключительно компоненту-обертке определяемому
 * внутри функции hoc, который генерирует
 * и устанавливает данные принадлежащие к типу
 * WrapperForWrappedProps [2] обертываемому-компоненту.
 */


                /**[3]      [4] */
export function withHoc<WrappedProps>(
        /**[5]              [6]         [7]                  [8] */
    WrappedComponent: ComponentType<WrappedProps &
WrapperForWrappedProps>)
       /**[9]    [10]           [11] */
        : FC<WrappedProps & WrapperProps>

/**
 * [3] определение универсальной функции hoc
 * чей единственный параметр типа WrappedProps [4]
 * представляет часть пропсов обертываемого-компонента, а их оставшаяся
часть, генерируемая
 * компонентом-оберткой определенным в теле hoc, к типу
WrapperForWrappedProps.
 *
 * Единственный параметр hoc WrappedComponent [5] принадлежит
 * к обобщенному типу Component<P> [6], которому в качестве аргумента
типа установлен
 * тип пересечение определяемый типами WrappedProps [7] и
WrapperForWrappedProps [8].
 *
 * Тип возвращаемого hoc значения обозначен как функциональный
компонент [9] который по мимо пропсов
 * устанавливаемых разработчиком и прокидываемых компонентом-оберткой
WrappedProps [10] ожидает ещё и пропсы
 * генерируемые и устанавливаемые компонентом-оберткой [11].
 *
 * [!] принадлежность возвращаемого hoc значения к функциональному типу
указана лишь по причине того
 *, что в нашем пример hoc возвращает именно его, а не классовый
компонент.
 */
```

Поскольку пример является минималистическим реализация тела *hoc* будет включать в себя лишь определение компонента-обертки выступающего в качестве возвращаемого значение. Тип компонента-обертки принадлежит к функциональному компоненту пропсы которого должны соответствовать типам описывающих как пропсы необходимые исключительно самому компоненту-обертке, так и оборачиваемому-компоненту. В теле компонента-обертки происходит разделение полученных пропсов на две части. Одна предназначается исключительно самому компоненту-обертке и служит для определения значений предназначенных для объединения со второй частью. Объединенные значения устанавливаются в качестве пропсов оборачиваемому-компоненту ссылка на который доступна через единственный параметр функции *hoc*. Стоит обратить внимание, что поскольку вторая часть пропсов образуется как остаточные параметры полученные при деструктуризации, то их тип принадлежит к типу `Pick<T, K>`, который для совместимости с типом описывающим прокидываемые компонентом-оберткой пропсы необходимо сначала привести к типу `unknown`, а уже затем к конкретному типу `WrappedProps`.

ts

```
import React, {ComponentType} from "react";


export interface WrapperProps {
    a: number;
    b: string;
}
export interface WrapperForWrappedProps {
    c: boolean;
}

export function withHoc<WrappedProps>(
    WrappedComponent: ComponentType<WrappedProps &
WrapperForWrappedProps>)
        : FC<WrappedProps & WrapperProps> {

                    /**[0]         [1]    [2]             [3] */
            const WrapperComponent:FC<WrappedProps & WrapperProps> =
props => {
                  /**[4]            [5] */
                let {a, b, ...wrappedOnlyProps} = props;
                        /**[6] */
                let wrapperToWrappedProps = {
                    c: true
                };
                       /**[7]
[8]                     [9]            [10]         [11] */
                let wrappedFullProps =
{...wrapperToWrappedProps, ...wrappedOnlyProps as unknown as
WrappedProps};

                            /**[12]               [13] */
                return <WrappedComponent {...wrappedFullProps} />
            }
```

```
                    /**[14] */
            return WrapperComponent;
}

/**
 * [0] определение компонента-обертки принадлежащего
 * к типу функционального компонента [1] пропсы которого
 * одновременно принадлежат к типам описывающих пропсы предназначаемые
 * исключительно обертываемому-компоненту WrappedProps [2] и
исключительно
 * компоненту-обертке WrapperProps [3]. В теле компонента-обертки общие
пропсы
 * разделяются с помощью механизма деструктуризации на две категории,
первая из
 * которых предназначается самому компоненту-обертке [4], а вторая
оборачиваемому-компоненту [5].
 * Поскольку пропсы предназначенные оборачиваемому-компоненту [5]
представляют из себя остаточные значения
 * полученные при деструктуризации, они принадлежат к типу Pick<T, K>,
что требует перед объединением их [9]
 * с пропсами созданными компонентом-оберткой [8] сначала к типу
unknown [10], а затем уже к необходимому
 * WrappedProps [11]. После этого слитые воедино пропсы можно
устанавливать [13] компоненту [12] ссылка на который
 * доступна в качестве единственного параметра hoc.
 *
 * [14] возвращаем из hoc компонент-обертку.
 */
```

Теперь необходимо определить компонент, пропсы которого будут принадлежать к типам описывающих значения, чья установка разделена между разработчиком и компонентом-оберткой. Далее этот компонент необходимо передать в качестве аргумента созданному нами *hoc*, чьё возвращаемое значение является компонентом-оберткой, с которым и будет взаимодействовать разработчик. Осталось создать экземпляр компонента-обертки и убедится как сила типизации позволяет установить в качестве пропсов только необходимые значения.

ts

```
/**[0] */
export interface CustomComponentProps {
    d: number;
    e: string;
}

                /**[1]                    [2]
[3]                [4] [5][5] */
export const CustomComponent: FC<CustomComponentProps &
WrapperForWrappedProps> = ({c, d, e}) => {
    return null;
}
```

```
                    /**[6]              [7]         [8] */
export const CustomComponentWrapped = withHoc(CustomComponent);

/**
 * [0] объявление типа CustomComponentProps представляющего пропсы
предназначенные
 * для обертываемого-компонента [1] и установка которых является
 * задачей разработчика. Пропсы компонента-обертки представленного
 * функциональным компонентом помимо типа CustomComponentProps [2]
 * описывающего значения устанавливаемые разработчиком [5]
 * также принадлежат к типу WrapperForWrappedProps [3] описывающего
 * значения устанавливаемые компонентом-оберткой [4].
 *
 * Ссылка на оборачиваемый-компонент передается в качестве аргумента [8]
 * функции hoc [7], а результат вызова сохраняется в переменную
представляющую
 * компонент-обертку [8].
 */


                    /**[9]   [9]    [10]  [10] */
<CustomComponentWrapped a={0} b={``} d={1} e={``} />; // Ok
                    /**[9]   [9]    [11]     [10]  [10] */
<CustomComponentWrapped a={0} b={``} c={true} d={1} e={``} />; // Error
-> Property 'c' does not exist on type CustomComponentProps &
WrapperProps'

/**
 * При создании экземпляра компонента-обертки будет необходимо
установить
 * параметры описываемые как типом WrapperProps [9] и так и
CustomComponentProps [10].
 * При попытке установить иные значения возникнет ошибка.
 */
```

[56.2] Определение hoc на основе классового компонента

Поскольку пример для *hoc*, возвращающего в качестве компонента-обертки классовый компонент, отличается от предыдущего лишь заменой функции на класс и объявлением для него двух дополнительных типов `*State` и `*Snapshot`, в повторном комментировании происходящего попросту нет смысла.

ts

```ts
import React, {ComponentType, Component} from "react";


export interface WrapperProps {
    a: number;
    b: string;
}

/**[0] */
interface WrapperState {}
/**[1] */
interface WrapperSnapshot {}

export interface WrapperForWrappedProps {
    c: boolean;
}


/**
 * Поскольку компонент-обертка будет представлен
 * классовым компонентом помимо описания его *Props
 * также появляется необходимость в объявлении типов
 * описывающих его *State [0] и *Snapshot [1].
 */

/**
 * [!] Стоит обратить внимание, что по причине
 * упрощенности примера отсутствуют более компактные
 * псевдонимы для менее компактных типов.
 */

export function withHoc<WrappedProps>(
    WrappedComponent: ComponentType<WrappedProps &
WrapperForWrappedProps>)
        : ComponentClass<WrappedProps & WrapperProps> {

                    /**[2] */
            class WrapperComponent extends Component<WrapperProps &
WrappedProps, WrapperState, WrapperSnapshot> {
                render() {
                    let {a, b, ...wrappedOnlyProps} = this.props;
                    let wrapperToWrappedProps = {
                        c: true
                    };
                    let wrappedFullProps =
{...wrapperToWrappedProps, ...wrappedOnlyProps as unknown as
WrappedProps};


                    return <WrappedComponent {...wrappedFullProps} />
                }
```

```
            }


            return WrapperComponent;
}

/**
 * [2] определение компонента-обертки в виде классового компонента.
 */

export interface CustomComponentProps {
    d: number;
    e: string;
}

export const CustomComponent: FC<CustomComponentProps &
WrapperForWrappedProps> = ({c, d, e}) => {
    return null;
}

export const CustomComponentWrapped = withHoc(CustomComponent);


<CustomComponentWrapped a={0} b={``} d={1} e={``} />; // Ok
<CustomComponentWrapped a={0} b={``} c={true} d={1} e={``} />; // Error
-> Property 'c' does not exist on type CustomComponentProps &
WrapperProps'
```

Глава 57

Пространства имен (namespace) и модули (module)

Поскольку на данный момент времени применение данных механизма не имеет смысла, то данную главу стоит изучить только в качестве исторической справки или при возникновении вопросов при работе с устаревшим кодом.

[57.0] Namespace и module — предназначение

Начать рассмотрение механизмов пространства имен и модулей стоит с уточнения области их применения. Эти механизмы не предназначены для масштабных приложений, которые должны строится при помощи модулей и загружаться с применением модульных загрузчиков. В настоящее время их можно использовать при написании небольших скриптов, внедряемых непосредственно в *html* страницу при помощи тега `<script>`, либо в приложениях, которые по каким-либо причинам не могут использовать модульную систему.

[57.1] Namespace - определение

Пространство имен — это конструкция, которая объявляется при помощи ключевого слова `namespace` и представляется в коде обычным *JavaScript* объектом.

ts
```ts
namespace Identifier {

}
```

Механизм пространства имен является решением такой проблемы, как коллизии в глобальном пространстве имен, дошедшего до наших дней из тех времён, когда ещё в спецификации *ECMAScript* не было определено такое понятие как модули. Простыми словами пространства имен — это совокупность обычной глобальной переменной и безымянного функционального выражения.

Объявленные внутри пространства имен конструкции срываются в безымянном функциональном выражении. Видимые снаружи конструкции записываются в объект, ссылка на который была сохранена в глобальную переменную переданную в качестве аргумента. Что записывать в глобальный объект, а, что нет, компилятору указывают при помощи ключевого слова `export`, о котором речь пойдет совсем скоро.

ts
```ts
// @info: До компиляции

namespace NamespaceIdentifier {
  class PrivateClassIdentifier {}
  export class PublicClassIdentifier{}
}
```

ts
```ts
// @info: После компиляции

var NamespaceIdentifier;

(function (NamespaceIdentifier) {

  class PrivateClassIdentifier {
  }
  class PublicClassIdentifier {
  }
```

```
  NamespaceIdentifier.PublicClassIdentifier = PublicClassIdentifier;

})(NamespaceIdentifier || (NamespaceIdentifier = {}));
```

Также стоит добавить, что `namespace` является глобальным объявлением. Это дословно означает, что пространство имен, объявленное как глобальное, не нуждается в экспортировании и импортировании, а ссылка на него доступна в любой точке программы.

[57.2] Модули (export, import) — определение

Модули в *TypeScript* определяются с помощью ключевых слов `export` / `import` и представляют механизм определения связей между модулями. Данный механизм является внутренним исключительно для *TypeScript* и не имеет никакого отношения к модулям `es2015` . В остальном они идентичны `es2015` модулям, за исключением определения модуля по умолчанию (*export default*).

ts

```ts
// Файл declaration.ts


export type T1 = {};

export class T2 {}
export class T3 {}

export interface IT4 {}

export function f1(){}

export const v1 = 'v1';
export let v2 = 'v2';
export var v3 = 'v3';
```

ts

```ts
Файл index.js


import {T2} from './declaration';
import * as declarations from './declaration';
```

Кроме того, объявить с использованием ключевого слова `export` можно даже `namespace`. Это ограничит его глобальную область видимости и его использование в других файлах станет возможным только после явного импортирования.

ts

```ts
// Файл declaration.ts


export namespace Bird {
  export class Raven {}
  export class Owl {}
}
```

ts

```ts
// Файл index.js


import {Bird} from "./declaration";

const birdAll = [ Bird.Raven, Bird.Owl ];
```

Необходимо отметить, что экспортировать `namespace` стоит только тогда, когда он объявлен в теле другого `namespace`, но при этом до него нужно добраться из программы.

ts

```ts
namespace NS1 {
  export namespace NS2 {
      export class T1 {}
  }
}
```

[57.3] Конфигурирование проекта

Для закрепления пройденного будет не лишним взглянуть на конфигурирование минимального проекта.

ts

```ts
// @info: Структура проекта
```

```
* /
   * dest
   * src
      * Raven.ts
      * Owl.ts
      * index.js
   * package.json
   * tsconfig.json
```

ts

```ts
// @filename: Raven.ts


namespace Bird {
  export class Owl {}
}
```

ts

```ts
// @filename: Owl.ts


namespace Bird {
  export class Raven {}
}
```

ts

```ts
// @filename: index.js


namespace App {
  const {Raven, Owl} = Bird;

  const birdAll = [Raven, Owl];

  birdAll.forEach( item => console.log(item) );
}
```

ts

```ts
// @filename: tsconfig.json


{
  "compilerOptions": {
      "target": "es2015",
      "module": "none",
      "rootDir": "./src",
      "outFile": "./dest/index.bundle.js"
  }
}
```

ts

```ts
// @filename: package.json


{
"name": "namespaces-and-modules",
"version": "1.0.0",
"description": "training project",
"main": "index.js",
"scripts": {
  "start": "./node_modules/.bin/tsc --watch",
  "build": "./node_modules/.bin/tsc"
},
"author": "",
"license": "ISC",
"devDependencies": {
  "typescript": "^2.5.2"
}
}
```

Осталось собрать проект, выполнив в консоли следующую команду:

sh

```sh
npm run build
```

Если все было сделано правильно, то в директории *dest* должен появится файл *index.bundle.js*.

ts

```ts
// @filename: index.bundle.js


var Bird;

(function (Bird) {
  class Owl {
  }
  Bird.Owl = Owl;
})(Bird || (Bird = {}));

var Bird;

(function (Bird) {
  class Raven {
  }
  Bird.Raven = Raven;
})(Bird || (Bird = {}));

var App;
```

```
(function (App) {
  const { Raven, Owl } = Bird;
  const birdAll = [Raven, Owl];
  birdAll.forEach(item => console.log(item));
})(App || (App = {}));
```

Глава 58

Настройка рабочего окружения

Иногда может потребоваться выполнить компиляцию *TypeScript* кода с помощью одного компилятор *tsc*. Именно поэтому текущая глава посвящена его установке, конфигурированию и запуску. Кроме того, она расскажет как спрятать длинные команды, изобилующие различными флагами компилятора, за коротенькими, определенными в *package.json*.

[58.0] Настройка рабочего окружения

Важным фактом является то, что насколько бы ни была продуктивной работа создателей *TypeScript*, им не успеть за развитием всей индустрии, всего сообщества. Простыми словами, насколько бы ни был продвинут компилятор, на практике его возможностей не хватает. Для того, что бы покрыть все потребности, разработчикам приходится прибегать к использованию сторонних библиотек, распространяемых через пакетный менеджер *npm*.

Кроме того, *html* и *css* используют в чистом виде, по большей части только в образовательных целях. В реальных проектах используют их более продвинутые аналоги, как например *jade* или *sass*, которые так же, как *TypeScript*, нуждаются в компиляторах. Также приложения не обходятся без шрифтов, иконок и изображений, которые в целях оптимизации принято предварительно обрабатывать. Поэтому современный процесс разработки не представляется возможным без специализированных сборщиков, таких как *webpack* или *gulp*.

gulp относится к так называемым *task runner'ам*, использование которых требует императивного определения задач для каждого отдельного процесса, самостоятельной настройки отлова ошибок. В свою очередь *webpack* — это настоящий комбайн, конфигурирование которого больше напоминает декларативный стиль. Но поскольку книга посвящена языку *TypeScript*, текущая глава будет ограниченна рассмотрением сборки проекта при помощи одного компилятора *tsc*.

[58.1] Сборка проекта с помощью tsc (TypeScript compiler)

Первым делом нужно создать директорию, в данном случае это будет директория с названием *typescript-with-tsc* содержащая две поддиректории *src* и *dest*. В первой будут находиться исходные файлы с расширением *.ts*, которые будут преобразованы в файлы с расширением *.js* и помещены во вторую директорию.

Теперь нужно открыть консоль в рабочей директории и выполнить инициализацию *npm*, в данном случае — ускоренную.

ts

```
npm init -y
```

На этот момент в директории должен появится файл *package.json*. После инициализации *npm*, установим компилятор *TypeScript*, выполнив в консоли следующую команду:

ts

```
npm i -D typescript
```

После успешной установки прежде всего нужно выполнить конфигурирование *TypeScript*. Для этого следовало бы выполнить в консоли:

ts

```
tsc init
```

Но, так как *TypeScript* установлен только локально, следует указать путь к нему:

ts

```
./node_modules/.bin/tsc --init
```

После этого в директории должен появится файл *tsconfig.json*, точную настройку которого можно произвести после прочтения главы, посвященной опциям компилятора (глава “Опции компилятора”), а пока просто укажем нужные настройки. В сгенерированном файле *tsconfig.json* будет очень много опций, большинство из которых закомментировано, но в итоге должно получиться не, что подобное:

ts

```
{
"compilerOptions": {
  "target": "es5",
   "module": "system",
   "outFile": "./dest/build.js",
   "rootDir": "./src"
},
"exclude": [
  "/node_modules/"
]
}
```

Теперь можно приступить к *dev* сборке. Для этого нужно открыть файл *package.json* и в поле `script` прописать команды для пакетного менеджера *npm*.

ts

```
{
"name": "typescript-with-tsc",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
  "build": "./node_modules/.bin/tsc --project ./tsconfig.json --watch",
  "test": "echo \"Error: no test specified\" && exit 1"
},
"author": "",
"license": "ISC",
"devDependencies": {
  "typescript": "^2.5.2"
}
}
```

Осталось только создать в директории *src* файл *index.ts* и запустить процесс разработки, выполнив в консоли команду:

ts

```
npm run build
```

После этого в папке *dest* должен появится скомпилированный *index.js*, а при изменении файлов в директории *src* преобразование должно запускаться автоматически. Сразу стоит обратить внимание на то, как именно компилятор понимает, какие файлы компилировать.

Для примера, создадим в директории *src* файл *hello-world.ts*, в котором объявим функцию, возвращающее приветствие.

ts
```ts
// Файл hello-world.ts


export function getMassage(): string {
  return 'Hello World!';
}
```

Важный момент заключается в том, что компилятор не будет обращать на этот файл внимание, пока он не будет задействован в программе, то есть не будет импортировать в `index.js`

ts
```ts
import {getMassage} from './hello-world';

console.log(getMassage()); // Hello World!
```

Такое поведение называется *Tree Shaking* и если по каким-либо причинам его нужно переопределить, то для этого нужно поправить конфигурацию компилятора, определив параметр `includes` представляющий массив ссылок на файлы которые необходимо компилировать независимо от их использования проектом.

ts
```ts
{
"compilerOptions": {
  "target": "es5",
  "module": "system",
  "outFile": "./dest/build.js",
  "rootDir": "./src/"
  },
  "include": [
    "./src/__/*.ts"
  ],
  "exclude": [
    "/node_modules/"
  ]
}
```

Очень часто бывает так, что при разработке в коде используются библиотеки, которых не должно быть в конечной сборке. Можно было бы каждый раз переписывать конфигурационный файл *tsconfig.json*, но есть способ сделать это элегантнее. Разделение *dev* сборки от *prod* осуществляется путем создания ещё одного конфигурационного файла. Назовем его *tsconfig.prod.json* и поместим в корне проекта. Стоит добавить, что конфигурационные файлы можно размещать где угодно, главное при запуске компилятора указать путь к нужному конфигу с помощью опции `--project`. Если это не сделать, компилятор будет искать файл *tsconfig.json* в той директории из под которой он был запущен.

sh

```
tsc -b --project ./tsconfig.json
```

Или...

sh

```
tsc -b ./tsconfig.json
```

Или...

sh

```
tsc -b --project ./tsconfig.props.json
```

Или...

sh

```
tsc -b ./tsconfig.props.json
```

После того, как конфигурационный файл был создан и отредактирован требуемым образом, остается только создать команду для запуска *prod* сборки. Для этого снова откройте файл *package.json* и в свойстве `script` укажите команду запуска компиляции, только на этот раз укажите путь до *tsconfig.prod.json*. Единственное, на, что стоит обратить внимание, при финальной сборке не нужно указывать опцию `--watch`, которая заставляет компилятор отслеживать изменения в файлах и автоматически перезапускать сборку.

ts

```
{
"name": "typescript-with-tsc",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
  "build": "./node_modules/.bin/tsc --project ./tsconfig.json --watch",
```

```
  "build:prod": "./node_modules/.bin/tsc --project ./
tsconfig.prod.json",
  "test": "echo \"Error: no test specified\" && exit 1"
},
"author": "",
"license": "ISC",
"devDependencies": {
  "@types/react": "^16.0.5",
  "@types/react-dom": "^15.5.4",
  "react": "^15.6.1",
  "react-dom": "^15.6.1",
  "typescript": "^2.5.2"
}
}
```

Чтобы запустить сборку, нужно, как и прежде, выполнить команду в терминале, только на этот раз указать другое имя.

sh

```
npm run build:prod
```

Также не будет лишним упомянуть, что реальные проекты практически всегда изобилуют множеством конфигурационных файлов. Поэтому если у Вас возникает мысль, что один конфигурационный файл не удовлетворяет условиям нескольких сборок, даже не раздумывайте, создавайте отдельный конфигурационный файл. При этом не отбрасывайте вариант с расширением одного конфигурационного файла другим с помощью свойства `extends`, более подробно о котором можно узнать из главы посвященной опциям компилятора.

Глава 59

Сборка с использованием ссылок на проекты

TypeScript реализует возможность, благодаря которой можно более эффективно работать одновременно над множеством проектов. Данный механизм значительно упростил работу с популярными в последнее время *монорепозиториями* — проектами, состоящими из отдельных, зависимых друг от друга и логически схожих самостоятельных частей. И хотя в данном механизме нет ничего сложного, ему рассмотрению будет посвящена вся текущая глава.

[59.0] Ссылки на проекты

Ссылки на проекты, путем объединения множества зависимых частей в общий процесс сборки, позволяют значительно сократить время сборки проекта. В этом компилятору `tsc` помогают алгоритмы интеллектуальных инкрементальных сборок, позволяющие компилировать только нуждающиеся в этом части. При этом под словом «часть» подразумевается не только фрагмент программы, но и отдельные проекты (модули). Кроме этого, ссылки на проекты после сборки повторяют исходную структуру связанных зависимостей, что немаловажно при работе с разбитыми на части проектами. Другими словами, ссылки на проекты позволяют компилировать множество отдельных проектов по правилам описанных, в уникальных для каждого проекта, `tsconfig.json`. В результате этого будет получено множество завершенных частей программы (библиотеки, динамически подгружаемые части приложения, микросервисы). Добиться подобного без использования ссылок на проекты невозможно, поскольку результатом

обычной компиляции является только одна часть программы, собранная по правилам одного конфигурационного файла.

Проект, который использует другие проекты в качестве своих зависимостей, должен указать ссылки на них в своем конфигурационном файле. Для этих целей в `tsconfig.json` существует специальное свойство верхнего уровня `references`. Свойство `references` ассоциируется с массивом объектов, поля которого описывают установки для связываемых проектов. Одним из ключевых полей является `path`, в качестве значения которого указывается ссылка на конфигурационный файл `tsconfig.json` проекта-зависимости, либо директорию, которая его содержит.

json

```
{
 "references": [
   { "path": "path/to/tsconfig.json" },
   { "path": "path/to/dir/with/tsconfig-json/" }
 ]
}
```

Важной деталью является то, что конфигурационный файл проекта, выступающего в качестве зависимости, обязан уведомлять об этом компилятор явным образом при помощи флага `composite`, установленного в `true`.

json

```
{
 "compilerOptions": {
   "composite": true
 }
}
```

Данный флаг гарантирует активацию других параметров, необходимых для более эффективной и быстрой работы поиска зависимостей. Активация флага `composite` переопределяет часть стандартного поведения. Например, если параметр `rootDir` не установлен явно, то он будет ссылаться на директорию, содержащую конфигурационный файл `tsconfig.json`, хотя обычно ссылается на директорию из которой был запущен процесс компиляции.

При разработке с использованием загружаемых через `npm` сторонних библиотек, выступающих в качестве источника информации о типах, применяются поставляемые в комплекте файлы декларации `.d.ts`. Разработчики могут быть в них уверены, так как после загрузки исходный код библиотек не изменяется. Когда в качестве зависимости выступает ссылка на проект, то изменение исходного кода является нормальной практикой. По этой причине, в качестве источников информации о типах используются сами файлы `.ts` или `.tsx`, составляющие эти проекты, поскольку только таким образом можно быть уверенным, что работа происходит с актуальной версией.

Если исходный код проектов, выступающих в качестве зависимостей, не подвержен частым изменениям, то сборку проекта можно ускорить за счет установки значения `fal`

se опции компилятора disableSourceOfProjectReferenceRedirect , что уведомит компилятор об исключении из наблюдения файлов проекта и указания в качестве источников типов файлов деклараций .d.ts . Данная опция указывается в конфигурационном файле проекта, выступающего в качестве зависимости. При этом нужно не забыть активировать саму генерацию деклараций, установив флаг declaration в true .

ts

```
{
 "compilerOptions": {
   "disableSourceOfProjectReferenceRedirect": true,
   "declaration": true,
   "composite": true,

   "outDir": "path/to/outDir/"
 }
```

Это ускорит процесс сборки и разработки, но потребует дополнительных усилий от разработчиков на пре-компиляцию необходимую для получения .d.ts . Кроме того, процесс сборки потребуется повторять при каждом изменении исходного кода проекта. Поэтому необходимость в опции disableSourceOfProjectReferenceRedirect кажется разумной лишь на очень больших проектах.

Разработчикам плагинов для сборщиков проектов будет не лишним узнать, что *TypeScript* реализует возможность позволяющую сторонним инструментам взаимодействовать с данным механизмом при помощи специального *api*.

Механизм ссылок на проекты неоценим на крупных проектах, а для случаев работы над небольшими проектами, существует флаг компилятора --build . Команда компилятора --build ожидает перечисление конфигурационных файлов и может быть использована совместно с такими специфичными для данной команды флагами, как --verbose , --dry , --clean и --force .

Флаг --verbose выводит подробную информацию о сборке и может сочетаться с любыми другими флагами. Флаг --dry выполняет сборку, не порождая выходные файлы. Флаг --clean удаляет выходные файлы, соответствующие заданным входным. --force принудительно выполняет не инкрементальную сборку. Кроме того, команда --build может сочетаться с такой командой как --watch , которая из всех специфичных для --build флагов сочетается только с флагом --verbose .

Не лишним будет упомянуть, что при использовании команды --build компилятор ведет себя так, будто опция --noEmitOnError установлена в true . Причина этому инкрементальная сборка, которая при наличии ошибки в коде, выводит уведомления только при непосредственной компиляции возникающей исключительно при условии внесения изменения в компилируемый граф зависимостей, что может привести к трудновыявляемым багам. Ведь в случае возникновения множества ошибок одновременно, разработчик первым делом возьмется за устранение ошибок возникших в других зависимостях, об ошибках в коде которого изменения не коснулись, компилятор

сообщать уже не будет. Но несмотря на вывод сообщений об ошибках в консоль, они так и останутся в программе.

Глава 60

Декларации

Декларации это очень важная часть *TypeScript* благодаря которой магия статической типизации проецируется на динамический *JavaScript*. Поэтому декларациям будет посвящена вся данная глава, рекомендуемая тем, кто только собирается писать свою первую типизированную библиотеку, которую планируется распространять с помощью *npm* репозитория.

[60.00] Что такое декларация (Declaration)

Поскольку при разработке программ на *TypeScript* используются библиотеки написанные на *JavaScript*, компилятор *tsc*, чьей главной задачей является проверка типов, чувствует себя будто у него завязаны глаза. Несмотря на то, что с каждой новой версией вывод типов все лучше и лучше учится разбирать *JavaScript*, до идеала ещё далеко. Кроме того, разбор *JavaScript* кода добавляет нагрузку на процессор, драгоценного время которого при разработке современных приложений, порой и так не достаточно.

TypeScript решил эту проблему за счет подключения к проекту заранее сгенерированных им или создаваемых вручную разработчиками деклараций. Декларации размещаются в файлах с расширением `.d.ts` и состоят только из объявлений типов полностью повторяющих программу до момента компиляции при которой она была лишена всех признаков типизации.

ts

```
Файл Animal.ts
```

```ts
export default class Animal {
   public name: string = 'animal';

   public voice(): void {}
}
```

ts

```
Файл Animal.d.ts


declare module "Animal" {
   export default class Animal {
       name: string;
       voice(): void;
   }
}
```

Еще не забыты дни, когда для часто используемых библиотек приходилось писать декларации вручную, при чем они часто содержали ошибки. Кроме этого, декларации не успевали обновляться под постоянно развивающиеся библиотеки. Сейчас такие проблемы кажутся уже нереальными, но несмотря на это, до сих пор приходится прибегать к использованию менеджера деклараций, который был создан в те самые далекие времена.

[60.01] Установка деклараций с помощью @types

Если декларация распространяется отдельно от библиотеки, то она скорее всего, попадет в огромный репозиторий на *github* под названием *DefinitelyTyped* содержащий огромное количество деклараций. Чтобы было проще ориентироваться в этом множестве, помимо сайта "TypeSearch" выступающего в роли поисковика, был создан менеджер деклараций под названием *Typed*. Но о нем мы говорить не будем поскольку он применяется при работе с *TypeScript* версии меньше чем *v2.0*, поэтому речь пойдет о его развитии в образе команды пакетного менеджера *npm*, а именно *@types*.

Для того, что бы установить требующуюся декларацию, в терминале необходимо выполнить команду, часть которой состоит из директивы `@types`, после которой через косую черту `/` следует имя библиотеки.

sh

```
npm i -D @types/name
```

Для примера, воспользуемся проектом созданным в теме посвященной настройки рабочего окружения и для демонстрации работы директивы `@types` установим всем известную библиотеку *React*.

Первым делом установим саму библиотеку *React* выполнив в терминале, запущенным из-под директории проекта, следующую команду.

sh

```
npm i -D react
```

Открыв директорию `/node_modules/` можно убедиться, что библиотека *React* успешно установлена, поэтому сразу же попытаемся импортировать её в файл `index.js` расположенным в директории *src*, предварительно изменив его расширение на требуемое для работы с *React* — `.tsx` .

ts

```
// @filename: src/index.js


import React, {Component} from 'react'; // Error
```

Несмотря на установленную на предыдущем шаге библиотеку *React*, при попытке импортировать её модули возникла ошибка. Возникла она потому, что компилятору *TypeScript* ничего не известно о библиотеке *React*, поскольку декларация описывающая типы поставляется отдельно от неё. Чтобы *tsc* понял, что от него хотят, требуется дополнительно установить декларацию при помощи команды `@types` пакетного менеджера `npm` .

sh

```
npm i -D @types/react
```

Ошибка, возникающая при импорте модулей *React* исчезла, а если заглянуть в директорию `_/node_modules/` , то можно увидеть новую примечательную поддиректорию `/@types` предназначенную для хранения устанавливаемых с помощью опции `@types` декларации.

Но для полноты картины и этого недостаточно. Для того, что бы добавить наш компонент в dom-дерево, необходимо установить *ReactDOM*, который уже давно развивается отдельной библиотекой.

sh

```
npm i -D react-dom
```

Кроме того, нужно установить необходимую для работы с ним декларацию.

sh

```
npm i -D @types/react-dom
```

Осталось только активировать опцию `--jsx` в `_tsconfig.json` и скомпилировать проект, как это было показано ранее.

ts

```ts
import React, {Component} from 'react'; // Ok
import * as ReactDOM from 'react-dom'; // Ok


const HelloReact = () => <h1>Hello react!</h1>;

ReactDOM.render(
  <HelloReact />,
  document.querySelector('#root')
);
```

[60.02] Подготовка к созданию декларации

Помимо того, что декларацию можно написать руками, её также можно сгенерировать автоматически, при условии, что код написан на *TypeScript*. Для того, что бы *tsc* при компиляции генерировал декларации, нужно активировать опцию компилятора `--declaration` .

Будет не лишним напомнить, что декларацию нужно генерировать только тогда, когда библиотека полностью готова. Другими словами, активировать опцию `--declaration` нужно в конфигурационном файле *production* сборки. Кроме того, в декларации нуждается только код, который будет собран в подключаемую библиотеку. Поэтому точкой входа в библиотеку должен быть файл, который содержит только импорты нужных модулей. Но разработка библиотеки невозможна без её запуска, а значит и точки входа в которой будет создан и инициализирован её экземпляр. Поэтому, что бы избежать чувства «что-то пошло не так», необходимо помнить, что при создании библиотеки требующей декларацию, в проекте может быть несколько точек входа. Точкой входа самого компилятора, служит конфигурационный файл который ему был

установлен при запуске. Это означает, что если проект находится в директории `src`, то в декларации путь будет указан как `src/libName` вместо требуемого `lib`.

ts

```ts
// Ожидается

declare module "libName" {
  //...
}
```

ts

```ts
// Есть

declare module "src/libName" {
  //...
}
```

Это в свою очередь означает, что при импорте модулей придется учитывать лишнюю директорию.

ts

```ts
// Ожидается

import {libName} from 'libName';
```

ts

```ts
// Есть

import {libName} from 'src/libName';
```

Это проблему можно решить, разместив конфигурационный файл в директории исходного кода, в нашем случае это директория `src`. Кто-то не придаст этому значение, кому-то это может показаться неэстетичным. Поэтому при рассмотрении генерации деклараций с помощью *tsc*, конфигурационный файл будет лежать непосредственно в директории `src`. Но при рассмотрении генерации деклараций с помощью сторонних библиотек, будет освещен альтернативный вариант.

Но и это ещё не все. Представьте, что Вы создаете библиотеку *React*, которая в коде представляется одноимённым классом расположенном в файле `React.ts`. При этом модуль, который будет представлять вашу библиотеку должен называться `react`, что в свою очередь обязывает задать имя файлу являющегося точкой входа как `react.js. Ну и, что спросите вы? Если вы ещё не знаете ответ на этот вопрос, то будете удивлены, узнав, что существуют операционные системы, как, например, *Windows*, которые расценивают пути до файлов *React.ts* и *react.ts* идентичными. Простыми словами если в директории присутствует файл с идентификатором *Identifier*, то ОС просто не позволит создать одноимённый файл, даже если его символы будут отличаться регистром. Именно об этом и будет сказано в ошибке, возникающей когда *TypeScript* обнаружит одноимённые файлы

в одной директории. Кроме того, если ваша операционная система позволяет создавать файлы чьи идентификаторы отличаются только регистром, помните, что разработчик работающий с вами в одной команде не сможет даже установить проект себе на машину, если его операционная система работает по другим правилам.

Поэтому решений у этой проблемы, на самом деле, всего два. Задавать идентификаторы отличающиеся не только регистром. Или же размещать файлы таким образом, что бы их идентификаторы не пересекались в одной директории. Но при этом нужно помнить, что структура модулей также изменится.

И поскольку *TypeScript* является компилируемым языком, не будет лишним напомнить правила именования директории в которую будет компилироваться результат. В случае разработки приложения, директорию содержащую скомпилированный результат принято называть *dest* (сокращение от слова *destination*). При разработке внешней библиотеки или фреймворка, директорию для собранных файлов принято называть *dist* (сокращение от слова *distributive*).

[60.03] Разновидности деклараций

На самом деле это глава должна называться «разновидности библиотек», так как именно о них и пойдет речь. Дело в том, что совсем недавно вершиной хорошего тона считалось объединение всего кода в один файл. Это же правило соблюдалось и при создании библиотек. Но сейчас все кардинально поменялось, и дело вот в чем.

В мире *JavaScript* существует большое количество библиотек, чей размер по меркам клиентских приложений превышает разумный. При этом отказ от них будет означать, что вам самому придется тратить драгоценное время на реализацию части их функционала. Это побудило создателей сборщиков наделять свои творения механизмом называющимся *Tree Shaking*.

Tree Shaking — это механизм позволяющий включать в сборку исключительно используемый код. Простыми словами, данный механизм позволяет взять только используемую часть от всей библиотеки. В перспективе это должно быть спасением, но на деле оказалось не совсем так.

Дело в том, что на данный момент *Tree Shaking* работает только если библиотека разбита на множество модулей. К примеру такие именитые библиотеки, как *lodash* или *rxjs*, для каждой отдельной функции создают отдельную точку входа, что при их использовании позволят значительно сократить размер конечного кода. Обозначим подобные библиотеки, как библиотеки с множеством точек входа. Кроме того, существуют библиотеки сопоставимые с монолитом, поскольку при использовании их малой части в конечную сборку они попадают целиком. Обозначим такие библиотеки, как библиотеки с единственной точкой входа.

[60.04] Декларации и область видимости

Важным моментом при создании деклараций для библиотек является понимание того, как их трактует компилятор. Дело в том, что все доступные компилятору декларации находятся в общей для всех области видимости. Это означает, что они так же, как переменные, функции и классы способны затенять или другими словами, перекрывать друг друга. Кроме того, идентификатор файла не играет никакой роли, поскольку компилятор рассматривает только определение деклараций с помощью ключевого слова `declare`. Проще говоря, два файла имеющие отличные идентификаторы, но идентичные объявления, будут затенять друг друга.

ts

```ts
// Файл ./types/petAnimal.d.ts


declare module "Pig" { // Error
  export default class Pig {}
}
declare module "Goat" { // Error
  export default class Goat {}
}
declare module "petAnimal" { // Ok
  export { default as Pig } from "Pig";
  export { default as Goat } from "Goat";
}
```

ts

```ts
// Файл ./types/wildAnimal.d.ts


declare module "Pig" { // Error
  export default class Pig {}
}
declare module "Goat" { // Error
  export default class Goat {}
}
declare module "wildAnimal" { // Ok
  export { default as Pig } from "Pig";
  export { default as Goat } from "Goat";
}
```

ts

```
// Файл index.js

import Pig from ‘Pig’; // From which library should import module?
```

Погружение в область видимости стоит начать с понимания процесса компилятора стоящего за установлением принадлежности к декларации в случаях, когда она распространяется не через менеджер `@types`. Прежде всего компилятор ищет в файле `package.json` свойство `types` и при его отсутствии или пустом значении `“”` переходит к поиску файла `index.d.ts` в корне директории. Если свойство `types` ссылается на конкретную декларацию, то точкой входа считается она. В противном случае файл `index.d.ts`. Стоит учесть, что при разработке будет возможно только взаимодействовать с модулями подключенными к точке входа. Кроме того, ограничить область видимости можно при помощи конструкций объявленных при помощи ключевых слов `module` или `namespace`. Единственное о чем сейчас стоит упомянуть, что области определяемые обеими конструкциями нужно расценивать как обычные модули, поскольку они могут включать только одно объявление экспорта по умолчанию (`export default`).

Если не уделить должного внимания области видимости при создании деклараций для подключаемых библиотек у разработчиков которые будут использовать подобные декларации с другими декларациями имеющими идентичное определение, могут возникнуть ошибки на этапе компиляции. Решений у этой проблемы всего два — сокрытие определений и уточнение определений. Способ к которому стоит прибегнуть зависит от вида разрабатываемой библиотеки.

[60.05] Декларации для библиотек с одной точкой входа

В проекте созданном в теме посвященной настройке рабочего пространства, в директории `src` создайте две точки входа, одну для разработки `index.js`, а другую для *prod-версии*, имя которой должно соответствовать имени библиотеки, в нашем случае это будет `index.lib.ts`.

По умолчанию точкой входа, как *npm* пакета, так и декларации, является файл с именем `index`. Поэтому, если в проект библиотеки имеет несколько точек входа, то важно не забыть указать имя файла с помощью свойства `types` `package.json`. Если для сборки используется *webpack*, то будет значительно проще изменить имя на `index` во время компиляции.

Кроме того, создайте два файла: `IAnimal.ts` и `Zoo.ts`. Также в директории `/src` создайте директорию `/animal`, в которой будут размещены два файла: `Bird.ts` и `Fish.ts`. В итоге должна получится следующая структура:

ts

```
* /
   * src
      * utils
         * string-util.ts
      * animal
         * Bird.ts
         * Fish.ts
      * IAnimal.ts
      * Zoo.ts
      * index.js
      * index.lib.ts
      * tsconfig.prod.ts
```

ts

```ts
// Файл IAnimal.ts


export interface IAnimal {
 name: string;
}
```

ts

```ts
// Файл utils/string-util.ts


export function toString( text: string ): string {
  return `[object ${ text }]`;
}
```

ts

```ts
// Файл animals/Bird.ts


import { IAnimal } from "../IAnimal";
import * as StringUtil from "../utils/string-util"

export default class Bird implements IAnimal {
  constructor(readonly name: string) {};

  public toString(): string {
      return StringUtil.toString(this.constructor.name);
  }
}
```

ts

```ts
// Файл animals/Fish.ts


import { IAnimal } from "../IAnimal";
import * as StringUtil from "../utils/string-util"

export default class Fish implements IAnimal {
  constructor(readonly name: string) {};

  public toString(): string {
      return StringUtil.toString(this.constructor.name);
  }
}
```

ts

```ts
// Файл Zoo.ts


import { IAnimal } from "./IAnimal";

export default class Zoo {
 private animalAll: IAnimal[ ] = [ ];

 public get length(): number {
     return this.animalAll.length;
 }

 public add(animal: IAnimal): void {
     this.animalAll.push(animal);
 }
 public getAnimalByIndex(index: number): IAnimal {
     return this.animalAll[index];
 }
}
```

ts

```ts
// Файл index.js


import Bird from "./animals/Bird";
import Fish from "./animals/Fish";

import Zoo from './Zoo';


const zoo: Zoo = new Zoo();

zoo.add( new Bird('raven') );
zoo.add( new Fish('shark') );

for( let i = 0; i < zoo.length; i++ ){
```

```
 console.log( `Animal name: ${ zoo.getAnimalByIndex(i).name }.` );
}
```

ts

```ts
// Файл index.lib.ts


/** imports */

import { IAnimal } from "./IAnimal";
import ZooCollection from './Zoo';

/** re-exports */

export {IAnimal} from './IAnimal'; // type

export {default as Bird} from './animals/Bird'; // type
export {default as Fish} from './animals/Fish'; // type

export {default as Zoo} from './Zoo'; // type

export const zoo: Zoo = new Zoo(); // instance
```

В коде нет ничего необычного, поэтому комментариев не будет. Если кому-то содержимое файла `index.lib.ts` показалось необычным, то стоит отметить, что это обычный реэкспорт модулей *JavaScript*, который никакого отношения к *TypeScript* не имеет. Повторю, файл `index.lib.ts` является точкой входа создаваемой библиотеки, поэтому он должен экспортировать все то, что может потребоваться при работе с ней. Конкретно в этом случае экспортировать *utils* наружу не предполагается, поэтому они не были реэкспортированы.

Также стоит обратить внимание на конфигурационные файлы *TypeScript*, которые взаимно добавляют точки входа друг друга в исключение. Кроме того, конфигурационный файл *dev-сборки* исключает также конфигурационный файл *prod-сборки*.

ts

```ts
// Файл /src/tsconfig.prod.json


{
   "compilerOptions": {
     "target": "es6",
     "module": "umd",
     "rootDir": "./",
     "declaration": true
   },
   "exclude": [
     "/node_modules",
     "./index.js"
```

```
    ]
 }
```

ts

```ts
//  Файл /tsconfig.json


{
"compilerOptions": {
  "target": "es6",
  "module": "umd",
  "rootDir": "./src"
},
"exclude": [
  "/node_modules",
  "./src/index.lib.ts",
  "./src/tsconfig.prod.json"
]
}
```

ts

```ts
// Файл package.json


{
"name": "zoo",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
  "build": "./node_modules/.bin/tsc --project ./tsconfig.json --watch",
  "build:prod": "./node_modules/.bin/tsc --project ./src/
tsconfig.prod.json"
},
"author": "",
"license": "ISC",
"devDependencies": {
  "typescript": "^2.5.2"
}
}
```

Осталось только запустить *prod-сборку* и если все было сделано правильно, в директории *dist* появятся скомпилированные файлы с расширением `.js` (конечный код) и `.d.ts` (представляющие декларации необходимые для работы как самого компилятора *TypeScript*, так и автодополнения *ide*).

ts

```ts
// Файл IAnimal.d.ts
```

```ts
export interface IAnimal {
  name: string;
}
```

ts

```ts
// Файл utils/string-util.d.ts


export declare function toString(text: string): string;
```

ts

```ts
// Файл animals/Bird.d.ts


import { IAnimal } from "../IAnimal";
export default class Bird implements IAnimal {
  readonly name: string;
  constructor(name: string);
  toString(): string;
}
```

ts

```ts
// Файл animals/Fish.d.ts


import { IAnimal } from "../IAnimal";
export default class Fish implements IAnimal {
  readonly name: string;
  constructor(name: string);
  toString(): string;
}
```

ts

```ts
// Файл Zoo.d.ts


import { IAnimal } from "./IAnimal";
export default class Zoo {
  private animalAll;
  readonly length: number;
  add(animal: IAnimal): void;
  getAnimalByIndex(index: number): IAnimal;
}
```

ts

```ts
// Файл index.d.ts


import Zoo from './Zoo';
```

```ts
/** re-exports */
export { IAnimal } from './IAnimal';
export { default as Bird } from './animals/Bird';
export { default as Fish } from './animals/Fish';
export { default as Zoo } from './Zoo';
/** exports */
export declare const zoo: Zoo;
```

Также стоит сказать, что сгенерированная декларация не может рассматриваться как единоверная. Очень часто можно увидеть декларации собранные в одном файле и сгруппированные по логическим признакам с помощью `namespace` или так называемых `ghost module` .

ts

```ts
/**ghost module */

declare module Zoo {
  interface IAnimal {
      name: string;
  }

  class Bird implements IAnimal {
      readonly name: string;
      constructor(name: string);
      toString(): string;
  }
  class Fish implements IAnimal {
      readonly name: string;
      constructor(name: string);
      toString(): string;
  }

  class Zoo {
      private animalAll;
      readonly length: number;
      add(animal: IAnimal): void;
      getAnimalByIndex(index: number): IAnimal;
  }

  const zoo: Zoo;
}
```

ts

```ts
/** module */

declare module "zoo" {
  export = Zoo;
}
```

Судить, какой из этих вариантов лучше, я не возьмусь, так как на мой взгляд, в данный момент, они оба не являются исчерпывающими. Возможно в будущем появятся новые правила для создания деклараций или редакторы будут по другому обрабатывать эти.

Также стоит обратить внимание, что в случае компиляции при помощи *tsc*, если в конечной директории присутствуют файлы, чьи имена совпадают с именами генерируемых при компиляции файлов, несмотря на их замену, ошибка все равно возникнет. Другими словами, если процесс сборки запускается не в первый раз, то нужно удалить файлы оставшиеся от предыдущей компиляции.

[60.06] Декларации для библиотек с множеством точек входа

При разработке библиотеки имеющий множество самостоятельных частей более разумно создавать каждую часть в виде отдельной точки входа. Это позволит использующим её приложениям за счет подключения только необходимых частей минимизировать вес конечной сборки, что становится возможно благодаря механизму *Tree Shaking*.

Для этого рассмотрим проект состоящий из самодостаточного модуля `bird.ts`, который делает ре-экспорт модуля `Raven.ts`, а также самодостаточного модуля `fish.ts` реэкспортирующего модуль `Shark.ts`. Кроме этого оба модуля доступны в точке входа `index.lib.ts`.

ts

```
* /
   * src/
      * to-string-decorate.ts
      * to-error-decorate.ts
      * index.lib.ts
```

Стоит сказать, что конфигурационные файлы ничем не отличаются от рассмотренных в теме создания деклараций для библиотек с одной точкой входа, поэтому их описание будет опущено.

ts

```
// Файл to-string-decorate.ts


export function toStringDecorate( type: string ): string {
```

```
  return `[object ${ type }]`;
}
```

ts

```ts
//Файл to-error-decorate.ts


export function toErrorDecorate( message: string, id: number = 0 ):
string {
  return `error:${ id === 0 ? '' : id }, ${ message }.`;
}
```

ts

```ts
// Файл index.lib.ts


/** re-export */

export {toStringDecorate} from './to-string-decorate';
export {toErrorDecorate} from './to-error-decorate';
```

После компиляции проекта в директорию *dist* сгенерируются следующие файлы -

ts

```ts
// Файл to-string-decorate.d.ts


export declare function toStringDecorate(type: string): string;
```

ts

```ts
// Файл to-error-decorate.d.ts


export declare function toErrorDecorate(message: string, id?: number):
string;
```

ts

```ts
// Файл index.d.ts


/** re-export */

export {toStringDecorate} from './to-string-decorate';
export {toErrorDecorate} from './to-error-decorate';
```

Сразу следует сказать, что с подобным описанием декларация не будет правильно функционировать, поэтому её придется подправить руками до следующего вида.

ts

```ts
// Файл to-string-decorate.d.ts


export declare function toStringDecorate(type: string): string;

export as namespace stringDecorate;
```

ts

```ts
// Файл to-error-decorate.d.ts


export declare function toErrorDecorate(message: string, id?: number):
string;

export as namespace errorDecorate;
```

ts

```ts
// Файл index.d.ts


/// <reference path="./to-string-decorate.d.ts" />
/// <reference path="./to-error-decorate.d.ts" />

declare module "zoo" {
  export default {stringDecorate, errorDecorate};
}
```

Обычно, как отдельную часть принято экспортировать только самодостаточные модули, такие как функции или классы. Но кроме того могут потребоваться объекты содержащие константы или, что-то незначительное, без чего отдельный модуль не сможет функционировать. Если такие объекты используются всеми самостоятельными модулями, то их можно также вынести в отдельный самостоятельный модуль. В случае, когда самодостаточному модулю для полноценной работы требуются зависимости, которые больше никем не используются, то такой модуль нужно оформлять так же, как обычную точку входа. Другими словами он должен содержать ре-экспорт всего необходимого. А кроме того экспортировать все как глобальный `namespace` с помощью синтаксиса:

ts

```ts
export as namespace identifier
```

Данный синтаксис объединяет все объявленные экспорты в глобальное пространство имен с указанным идентификатором. Затем объявленные пространства имен нужно импортировать в точку входа с помощью директивы с тройным слешем `/// <reference path=””/>` , после чего экспортировать из объявленного модуля.

[60.07] Создание деклараций вручную

Описывать незначительные декларации самостоятельно (вручную) приходится довольно часто. Потребность возникает при необходимости задекларировать импортируемые файлами `.ts` нестандартные для него расширения файлов. Дело в том, что компилятор *TypeScript* понимает только импорт расширения `.ts` / `.tsx` / `.d.ts` / `.json`, а с активной опцией `--allowJS`, еще и `.js` / `.jsx`. Но работая с таким сборщиком как в *webpack* или используя *css-in-js*, придется импортировать в код файлы с таким расширением, как `.html`, `.css`, и т.д. В таких случаях приходится создавать декларации файлов вручную.

Самостоятельно объявление деклараций начинается с создания директории предназначенной для их хранения. В нашем случае это будет директория *types* расположенная в корне проекта. Декларации можно складывать прямо в неё, но будет правильно считается создавать под каждую декларацию отдельную поддиректорию носящую имя модуля нуждающегося в ней. Поэтому создадим поддиректорию с именем `/css`, а уже в ней создадим файл `index.d.ts`. Откроем этот файл и напишем в нем декларацию определяющую расширение `.css`.

ts

```ts
// Файл ./types/css/index.d.ts


declare module "*.css" {
  const content: any;
  export default content;
}
```

В тех случаях, когда модуль определяет тип `any`, более уместно использовать при объявлении сокращенный вариант, который предполагает тип `any`.

ts

```ts
declare module "*.css";
```

Осталось только подключить декларацию в конфигурационном файле и ошибок при импорте расширения `.css` не возникнет.

ts

```
// Файл tsconfig.json


{
  "compilerOptions": {
      "target": "es2015",
      "module": "none",
      "rootDir": "./src",
      "typeRoots": [
          "./types"
      ]
  },
  "exclude": [
      "./node_modules"
  ]
}
```

Будет не лишним упомянуть, что самостоятельное создание деклараций, помимо нестандартных расширений, также часто требуется при необходимости расширения типов описывающих внешние библиотеки. Например, если при работе с библиотекой *React* возникнет необходимость в использовании пользовательских свойств определенных спецификацией *html*, то придется расширять объявляемый в её модуле тип `HTMLAttributes`.

[60.08] Директива с тройным слешем (triple-slash directives)

До этого момента было рассмотрено создание библиотек представленных одним или больше количеством самостоятельных модулей. Акцент в этом предложении необходимо сделать на слове *самостоятельных*, поскольку они не были зависимы от каких-либо других модулей (деклараций). Если разрабатываемая библиотека представляет из себя множество зависящих друг от друга модулей или она зависит от деклараций устанавливаемых с помощью директивы `@types`, то генерируемые декларации также будут нуждаться в зависимостях. Для этих случаев существует директива `/// <reference types=""/>`. Данная директива указывается в начале файла и предназначена для подключения деклараций, путь до которых указывается с помощью атрибута `types`.

ts

```
/// <reference types="react" />
```

Кроме того, с помощью данной директивы можно указать версию используемой библиотеки.

ts

```
/// <reference lib="es2015" />
```

Подобный функционал будет полезен разработчикам деклараций `.d.ts`, которые зависят от конкретной версии *ECMAScript*.

[60.09] Импортирование декларации (import)

Помимо типов, описанных в глобальных декларациях, в аннотациях типов также можно использовать типы из деклараций импортированных с помощью директивы `import`.

ts

```ts
// file declaration-excluded-from-global-scope/animal.d.ts

export declare interface IAnimal {
  type: string;
}
```

ts

```ts
// file src/index.js

import * as DTS from "declaration-excluded-from-global-scope/animal";

// импорт декларации на уровне модуля

let v0: DTS.IAnimal = { type: '' }; // Ok
let v1: DTS.IAnimal = { type: 5 }; // Error


// инлайн импорт

let v2: import('declaration-excluded-from-global-scope/animal').IAnimal
= { type: '' }; // Ok
```

```ts
let v3: import('declaration-excluded-from-global-scope/animal').IAnimal
= { type: 5 }; // Error
```

Этот механизм также позволяет указывать аннотацию типов непосредственно в файлах с расширением `.js` .

ts

```ts
// file declaration-excluded-from-global-scope/animal.d.ts

export declare interface IAnimal {
  type: string;
}
```

ts

```ts
// file lib/index.js

/**
*
* @param {import("./declaration-excluded-from-global-scope/
animal").IAnimal} animal
*/
export function printAnimalInfo(animal){ animal.type; // autocomplete }
```

ts

```ts
// file src/index.js

import * as AnimalUtils from "lib/index.js";


AnimalUtils.printAnimalInfo( { type: '' } ); // Ok
AnimalUtils.printAnimalInfo( { type: 5 } ); // Error
```

Глава 61

Публикация TypeScript

В командах невозможно вести разработку приложений без вынесения некоторых частей кода в отдельные *npm* пакеты. При этом модули использующие *TypeScript* требуют особой конфигурации, которой и посвящена данная глава.

[61.0] Публикация

Для того, что бы подключать создаваемые декларации `.d.ts*` с помощью пакетного менеджера *npm*, их нужно определить как *npm-пакет* (npm package). В дальнейшем этот пакет можно опубликовать с помощью того же пакетного менеджера *npm* в npm-репозитории, который может быть как публичным, доступным всем разработчикам, так и приватным, доступным только автору/команде. Кроме того, можно и вовсе обойтись без публикации в репозиторий, подключая пакет с локального диска. Сама публикация здесь рассматриваться не будет, с этим процессом можно ознакомится в документации к пакетному менеджеру *npm*, но не будет лишнем упомянуть о тонкостях, которые привнес *TypeScript*.

Говоря о создании npm-пакета, мы подразумеваем создание описания нашего кода в файле `package.json`. Для того, что бы компилятор *TypeScript* смог использовать декларацию `.d.ts`, ему нужно помочь, присвоив полю `types` путь до неё.

ts

```
// package.json

{
```

```
    "types": "path to file declaration .d.ts"
}
```

Помимо этого с помощью поля `typesVersion` можно указать декларацию в зависимости от версии *TypeScript*.

ts

```
// package.json

{
    "typesVersions": {
        ">=3.1": { "*": ["ts3.1/index.d.ts"] }
    }
}
```

В случае, если версия среды *TypeScript* не подпадает под указанный диапазон, то разрешение кода будет выполнено с помощью декларации указанной в поле `types` .

ts

```
// package.json

{

    "typesVersions": {
        ">=3.1": { "*": ["ts3.1/index.d.ts"] } // если версия TypeScript
выше либо равна 3.1
    }

    // если версия TypeScript ниже версии 3.1, то будет выбрана
декларация указана в поле types

      "types": "./dest/index.d.ts", // если версия TypeScript ниже
версии 3.1
}
```

Правила, разрешающие использование версий библиотек *TypeScript*, полностью идентичны правилам диапазонов версионирования *nodejs* ranges.

Кроме того, можно указывать несколько диапазонов одновременно. Но нужно помнить, что порядок указания диапазонов важен, так как они могут перекрывать друг друга.

ts

```
{
   "typesVersions": {
       ">=3.2": { "*": ["ts3.2/index.d.ts"] },
       ">=3.1": { "*": ["ts3.1/index.d.ts"] }
   }
}
```

Глава 62

Опции компилятора

Если сравнить компилятор `tsc` с фортепиано, то его опции сопоставимы с камертоном позволяющим настроить его наилучшим образом, что на практике означает сократить время сборки и повысить типобезопасность проекта.

[62.00] strict

`--strict` - активирует все флаги, входящие в группировку строгого режима и сопутствующие повышению типобезопасности программы. На данный момент флаг `strict` активирует следующие опции компилятора: `--strictNullChecks`, `--noImplicitAny`, `--noImplicitThis`, `--alwaysStrict`, `--strictFunctionTypes`, `--strictPropertyInitialization` и `--strictBindCallApply`. Несмотря на то, что флаг `strict` активирует сразу все указанные флаги, при желании конкретные флаги можно отключить.

ts

```
{
  "compilerOptions": {
      "strict": false,
      "strictNullChecks": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.01] suppressExcessPropertyErrors

`--suppressExcessPropertyErrors` - если данная опция активна, то компилятор перестает проверять литералы объекта на излишние члены.

ts

```
{
  "compilerOptions": {
      "suppressExcessPropertyErrors": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

Пример

ts

```
interface T1 {
  f1: number;
}

let v1: T1 = {f1: 0, f2: ''}; // suppressExcessPropertyErrors ===
false ? Error : Ok
```

[62.02] suppressImplicitAnyIndexErrors

`--suppressImplicitAnyIndexErrors` - при активной опции `--noImplicitAny` подавляет ошибки, связанные с добавлением динамических свойств в объекты, у которых отсутствует индексное определение.

ts

```
{
  "compilerOptions": {
```

```
        "suppressImplicitAnyIndexErrors": false
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

Пример для неактивной опции

ts

```ts
// tsconfig.json

{
    "compilerOptions": {
        "noImplicitAny": true,
        "suppressImplicitAnyIndexErrors": false
    }
}
```

ts

```ts
// index.js

interface Member {

}
interface IndexMember {
    [key: string]: any;
}


let memberObject: Member = {};
memberObject['name'] = 'object'; // Error

let indexMemberObject: IndexMember = {};
indexMemberObject['name'] = 'object'; // Ok
```

Пример для активной опции

ts

```ts
// tsconfig.json

{
    "compilerOptions": {
        "noImplicitAny": true,
        "suppressImplicitAnyIndexErrors": false
    }
}
```

ts

```
// index.js

interface Member {

}
interface IndexMember {
  [key: string]: any;
}


let memberObject: Member = {};
memberObject['name'] = 'object'; // Ok

let indexMemberObject: IndexMember = {};
indexMemberObject['name'] = 'object'; // Ok
```

[62.03] noImplicitAny

`--noImplicitAny` - при активной опции выводит ошибку, если вывод типов установил принадлежность типа члена к `any` .

ts

```
{
  "compilerOptions": {
      "noImplicitAny": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

Пример с неактивной опцией

ts

```
let v1; // Ok

function f1(p1) { // Ok
  let v1; // Ok

  return p1;
}
```

```
type T2 = {f1}; // Ok

interface IT1 {
  f1; // Ok
}

class T1 {
  public f1; // Ok
}
```

Пример с активной опцией

ts

```
let v1; // Ok

function f1(p1) { // Parameter 'p1' implicitly has an 'any' type.
  let v1; // Ok

  return p1;
}

type T2 = {f1}; // Member 'f1' implicitly has an 'any' type

interface IT1 {
  f1; // Member 'f1' implicitly has an 'any' type
}

class T1 {
  public f1; // Member 'f1' implicitly has an 'any' type
}
```

[62.04] checkJs

`--checkJs` - данная опция говорит компилятору, что код, который находится в файлах с расширением `.js`, также нужно проверять на ошибки. При этом можно исключить определенные файлы из проверки, добавив им строку `// @ts-nocheck`. Или наоборот, можно попросить компилятор проверить только помеченные как `// @ts-check` файлы без активации опции `--checkJs`. К тому же можно игнорировать ошибки на конкретных строках, указав `// @ts-ignore: error message` предыдущей строке.

ts

```
{
  "compilerOptions": {
      "checkJs": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

Кроме этого, при активной опции `--checkJS` компилятору с помощью аннотации `/** @type {...} */` можно указывать типы прямо в файлах с расширением `.js` .

При использовании *JavaScript* в *TypeScript* коде нарушается типобезопасность программы.

ts

```ts
// file sum.js

export function sum(a,b){
  return a + b;
}
```

ts

```ts
// file index.js

import {sum} from "./sum.js";

let n = sum('5', 5); // len n: any, кроме того результаты выполнения функции будет неверным
```

Но это можно исправить с помощью специальной аннотации, которая располагается в *JavaScript* коде.

ts

```ts
// file sum.js

/** @type {(a:number,b:number) => number)} */
export function sum( a,b){
  return a + b;
}
```

ts

```
// file index.js

import {sum} from "./sum.js";

let n0 = sum('5', 5); // Error
let n1 = sum(5, 5); // Ok, let n1: number
```

[62.05] JSX

`--jsx` - данная опция указывает компилятору, какое расширение указывать `.tsx` файлам после компиляции. Все дело в том, что у *React* существует два вида приложений, одни создаются для веб-платформы, другие для мобильных платформ. Кроме того, файлы для веб-платформы на входе должны иметь расширение `.jsx`, в, то время как для мобильной платформы — `.js`. Поэтому компилятору нужно указывать, в какой именно формат преобразовывать файлы с расширением `.tsx`.

При указании `"react"` в качестве значения опции `--jsx` компилятор преобразует `.tsx` в `.jsx` файлы, которые затем компилируются в `.js`. Если в качестве значения будет указано `"preserve"`, то компилятор преобразует `.tsx` в `.jsx`, которые сохраняют *XML-подобный* синтаксис. Если указать значение `"react-native"`, то компилятор преобразует файлы `.tsx` в требующийся `.js`.

ts

```
{
  "compilerOptions": {
      "jsx": "react" | "react-native" | "preserve"
  }
}
```

type: `string` **default**: `preserve` **values**: `react`, `react-native`, `preserve`

[62.06] jsxFactory

`--jsxFactory` - данная опция позволяет при трансляции файлов `.tsx` в `.js` переопределить фабрику рендера.

ts

```ts
// default

// from file .tsx

import * as React from "react";

<h1>Ok</h1>
```

ts

```ts
// to file .js

"use strict";
exports.__esModule = true;
var React = require("react");
React.createElement("h1", null, "Ok");
```

Установив желаемое значение текущей опции появляется возможность переопределить функцию рендера *React* на любую другую.

ts

```ts
// set options jsxFactory to dom

// from file .tsx

import dom from "dom";

<h1>Ok</h1>
```

ts

```ts
// set options jsxFactory to dom

// to file .js

"use strict";
```

```
/** @jsx renderer */
exports.__esModule = true;
var dom = require("dom");
dom["default"]("h1", null, "Ok");
```

Кроме того, подобного поведения можно добиться при помощи нотации `/** @jsx identifier */`, которая указывается в начале файла, а вместо *identifier* вписывается имя функции рендера.

ts

```ts
// from file .tsx

/** @jsx renderer */

import renderer from "renderer";

<h1>Ok</h1>
```

ts

```ts
// to file .js

"use strict";
/** @jsx renderer */
exports.__esModule = true;
var renderer = require("renderer");
renderer["default"]("h1", null, "Ok");
```

Помимо этого, аннотация `/** jsx identifier */` позволяет переопределить функцию рендера, переопределенную с помощью опции `--jsxFactory`.

ts

```ts
{
  "compilerOptions": {
      "jsxFactory": "React.createElement"
  }
}
```

type: `string` **default**: `React.createElement` **values**: `"*"`

[62.07] target (t)

`--target`, или сокращенно `-t` - указывает компилятору, с какой версией спецификации должен быть совместим генерируемый *JavaScript* код. По умолчанию установлена совместимость с `ES3`. Кроме того, можно указывать совместимость с `ES5`, `ES2015` (она же `ES6`), `ES2016`, `ES2017`, `ESNext`. Стоит добавить, что `ESNext` равноценно *latest version*.

ts

```
{
  "compilerOptions": {
        "target": "es3"
  }
}
```

type: `string` **default**: `es3` **values**: `es3`, `es5`, `es6` / `es2015`, `es2016`, `es2017`, `esnext`

[62.08] extends

`extends` - с помощью этого свойства можно расширять конфигурацию `tsconfig.json`.

ts

```
// tsconfig.base.json

{
  "compilerOptions": {
      "target": "es2015"
  }
}
```

ts

```
// tsconfig.json

{
  "extends": "./tsconfig.base.json"
}
```

type: `string` **default**: `""` **values**: `*`

Кроме того, при использовании механизма расширения (`extends`) поиск файла конфигурации `tsconfig.json` может осуществляться по пакетам (`packages`) `Node JS` модулей, находящихся в директории `node_modules` .

ts

```
// tsconfig.json

{
  "extends": "some-npm-module"
}

// or

{
  "extends": "some-npm-module/some-tsconfig.json"
}
```

Алгоритм разрешения аналогичен алгоритму поиска модулей самого `NodeJS` . Проще говоря, если путь, указанный в качестве значения атрибута `extends` не будет найден в директории `node_modules` , находящейся в текущей директории, то поиск продолжится в директории `node_modules` вверх по дереву. Но есть одна особенность, которая заключается в том, что при разрешении пути компилятор `tsc` , зайдя в директорию `node_modules` , сначала проверяет `package.json` на наличие атрибута `tsconfig` , которому в качестве значения указывают путь до конфигурационного файла. Если атрибут `tsconfig` найден, то конфигурация, на которую он ссылается, будет установлена в качестве расширяемой. Если `package.json` не содержит атрибут `tsconfig` , то в качестве расширяемого конфигурационного файла будет выбран файл `tsconfig.json` , находящийся в корне директории. Если в корне директории файла `tsconfig.json` найдено не будет, то поиск продолжится вверх по дереву.

ts

```
/project
    /node_modules
        /some-module
            package.json // with attr "tsconfig": "tsconfig.custom.json"
            tsconfig.custom.json
            tsconfig.json
    tsconfig.json // "extends": "some-module"

// в этом случае в качестве расширяемого конфигурационного файла будет
```

```
выбран файл находящийся по пути /node_modules/some-module/
tsconfig.custom.json
```

ts

```
/project
    /node_modules
        /some-module
            package.json // without attr "tsconfig"
            tsconfig.custom.json
            tsconfig.json
    tsconfig.json // "extends": "some-module"

// в этом случае в качестве расширяемого конфигурационного файла будет
выбран файл находящийся по пути /node_modules/some-module/tsconfig.json
```

ts

```
/project
    /node_modules
        /some-module
            package.json // without attr  "tsconfig"
            tsconfig.custom.json
            tsconfig.json
    tsconfig.json // "extends": "some-module/tsconfig.custom.json"

// в этом случае в качестве расширяемого конфигурационного файла будет
выбран файл находящийся по пути /node_modules/some-module/
tsconfig.custom.json
```

[62.09] alwaysStrict

`--alwaysStrict` - данная опция говорит компилятору, что рассматривать и генерировать код нужно с учетом строгого режима `“use strict”`.

ts

```
{
  "compilerOptions": {
      "alwaysStrict": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.10] strictNullChecks

`--strictNullChecks` - активировав эту опцию, компилятор не позволяет указывать в качестве значения типы `Null` и `Undefined`, если они не были указаны в аннотации.

ts

```
{
  "compilerOptions": {
      "strictNullChecks": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.11] stripInternal

`--stripInternal` - когда данная опция активна, компилятор не создает деклараций `.d.ts` для файлов, помеченных как `/** @internal */`

ts

```
{
  "compilerOptions": {
      "stripInternal": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.12] noImplicitThis

`--noImplicitThis` - в активном состоянии запрещает использование `this` в местах, не предусмотренных контекстом.

ts

```
{
  "compilerOptions": {
      "noImplicitThis": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

Пример с неактивной опцией

ts

```
function f1(){
  this.name = 'newName'; // Ok -> context T1 window
}
function f2(this: T1){
  this.name = 'newName'; // Ok -> context T1
}

this.name = 'newName'; // Error -> context window

class T1 {
  public name: string;

  public m1(name: string): void {
      this.name = name; // Ok -> context T1
  }
}
```

Пример с активной опцией

ts

```
function f1(){
  this.name = 'newName'; // Error -> context T1 window
}
```

```
function f2(this: T1){
  this.name = 'newName'; // Ok -> context T1
}

this.name = 'newName'; // Error -> context window

class T1 {
  public name: string;

  public m1(name: string): void {
      this.name = name; // Ok -> context T1
  }
}
```

[62.13] noImplicitUseStrict

`--noImplicitUseStrict` - при активной опции ошибки будут выводиться в случаях, при которых в поток компиляции попадут файлы, содержащие `'use strict'`. Кроме того, скомпилированные файлы также не будут содержать указание `'use strict'`.

ts

```
{
  "compilerOptions": {
      "noImplicitUseStrict": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.14] baseUrl

`--baseUrl` - указывает базовый путь, относительно которого будет производиться поиск модулей.

ts

```
{
  "compilerOptions": {
      "baseUrl": ""
  }
}
```

type: `string` **default**: `""` **values**: `base path`

ts

```ts
// m1 module ./node_modules/m1

// tsconfig.json  "baseUrl": "./node_modules"

// index.js

import M1 from 'm1';
```

[62.15] paths

`--paths` - с помощью этой опции создаются псевдонимы для используемых в программе модулей.

ts

```
{
  "compilerOptions": {
      "paths": {
          "name": [ “path/to/lib” ]
      }
  }
}
```

type: `object` **default**: `null` **values**: `{[key]: *}`

Пример

ts

```ts
// tsconfig.json
{
  "compilerOptions": {
      "paths": {
```

```
            "jquery": ["node_modules/jquery/dest/jquery.min.js"]
        }
    }
}

// or

// tsconfig.json
{
    "compilerOptions": {
        "baseUrl": "./node_modules",
        "paths": {
            "jquery": ["jquery/dest/jquery.min.js"]
        }
    }
}

import jquery from 'jquery';

// or

// tsconfig.json
{
    "compilerOptions": {
        "baseUrl": "./node_modules",
        "paths": {
            "jQ": ["jquery/dest/jquery.min.js"]
        }
    }
}

import jquery from 'jQ';
```

[62.16] rootDir

`--rootDir` - с помощью этой опции можно ограничить область выборки файлов для компиляции. В качестве значения выступает строка — путь до конкретной директории или файла.

ts

```
{
    "compilerOptions": {
        "rootDir": ""
```

```
  }
}
```

type: `string` **default**: `""` **values**: `path to dir with .ts files`

[62.17] rootDirs

`--rootDirs` - с помощью этой опции можно ограничить область выборки файлов для компиляции. В качестве значения принимается массив с путями до директорий и файлов.

ts

```
{
  "compilerOptions": {
      "rootDirs": [

      ]
  }
}
```

type: `array[ ]` **default**: `null` **values**: `path to dir with .ts files`

[62.18] traceResolution

`--traceResolution` - в случае активной текущей опции, при компиляции будет выводиться информация о собираемых модулях.

ts

```
{
  "compilerOptions": {
      "traceResolution": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.19] lib

`--lib` - с помощью этой опции можно управлять конструкциями, которые включены в ту или иную версию *ES*.

ts

```
{
  "compilerOptions": {
      "lib": [

      ]
  }
}
```

type: `string[ ]` **default**: for es5 `[dom, es5, ScriptHost]` , for es6 `[dom, es6, dom.Iterable, ScriptHost]` **values**: `dom` , `webworker` , `es5` , `es6` / `es2015` , `es2015.core` , `es2015.collection` , `es2015.iterable` , , `es2015.promise` , `es2015.proxy` , `es2015.reflect` , `es2015.generator` , `es2015.symbol` , `es2015.symbol.wellknown` , `es2016` , `es2016.array.include` , `es2017` , `es2017.object` , `es2017.sharedmemory` , `scripthost`

[62.20] noLib

`--noLib` - не использует декларацию `lib.d.ts` по умолчанию.

ts

```
{
  "compilerOptions": {
      "noLib": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.21] noResolve

`--noResolve` - данная опция говорит компилятору не компилировать файлы, которые не были указаны в командной строке.

ts

```ts
// terminal

tsc index.js T1 --noResolve
```

ts

```ts
// index.js

import T1 from './T1'; // Ok
import T2 from './T2'; // Error
```

[62.22] noStrictGenericChecks

`--noStrictGenericChecks` - отключает строгую проверку параметров типа для функциональных типов.

ts

```ts
{
  "compilerOptions": {
      "noStrictGenericChecks": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.23] preserveConstEnums

`--preserveConstEnums` - говорит компилятору не удалять из исходного кода перечисления (`enum`), объявленные как `const` .

ts

```
{
  "compilerOptions": {
      "preserveConstEnums": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

Пример с неактивной опцией

ts

```
// .ts

const enum Animal {
  Bird = 'bird',
  Fish = 'fish'
}

let greeting: string = `Hello ${ Animal.Bird }!`;
```

ts

```
// .js

let greeting = `Hello ${"bird" /* Bird */}!`;
```

Пример с активной опцией

ts

```
// .ts

const enum Animal {
  Bird = 'bird',
  Fish = 'fish'
}
```

```ts
let greeting: string = `Hello ${ Animal.Bird }!`;
```

ts

```ts
// .js

var Animal;
(function (Animal) {
  Animal["Bird"] = "bird";
  Animal["Fish"] = "fish";
})(Animal || (Animal = {}));
let greeting = `Hello ${"bird" /* Bird */}!`;
```

[62.24] removeComments

`--removeComments` - удаляет комментарии из сгенерированного `.js` .

ts

```ts
{
  "compilerOptions": {
      "removeComments": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.25] noUnusedLocals

`--noUnusedLocals` - активная опция заставляет компилятор выводить сообщения о неиспользуемых элементах кода. Простыми словами, если в коде что-то объявлено, но не используется, будет возникать ошибка.

ts

```
{
  "compilerOptions": {
      "noUnusedLocals": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

Пример

ts

```ts
import T1 from "./T1"; // Error

let v0: number; // Warning
let v1: number = 0; // Warning

function f /** Warning */ (  ){
  let lv0: number; // Warning
  let lv1: string = 0; // Error

  f();
}

class C /**Error */ {
  private f0: number; // Warning
  private f1: number; // Warning
  private f2: number = 0; // Warning

  constructor(  ) {
      this.f0 = 0;

      let c: C = new C(); // Warning
  }

  private m0 /** Warning */ (){} // Warning
  private m1 /** Warning */ (){  // Warning
      this.m1();
  }
}
```

[62.26] noUnusedParameters

`--noUnusedParameters` - данная опция заставляет компилятор выводить ошибки, если в коде будут найдены функции, чьи параметры не используются (за исключением параметров идентификаторы которых начинаются с нижней черты, например, `_prop`).

ts

```
{
  "compilerOptions": {
      "noUnusedParameters": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

Пример

ts

```
function f ( p0: number /** Warning */,
          p1: number /** Warning */,
          p2: number = 0 /** Warning */ ){
  p1 = 0;
}

class C {
  constructor( p0: number /** Warning */,
      p1: number /** Warning */,
      p2: number = 0 /** Warning */ ) {}

  private m ( p0: number /** Warning */,
      p1: number /** Warning */,
      p2: number = 0 /** Warning */ ){
          p1 = 0;
      }
}
```

[62.27] skipLibCheck

`--skipLibCheck` - при активной опции, компилятор перестает проверять типы в файлах библиотеках с расширением `.d.ts`, что экономит время, но может привести к редким ошибкам, связанным с типами.

ts

```
{
  "compilerOptions": {
      "skipLibCheck": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.28] declarationDir

`--declarationDir` - указывает директорию, откуда будут подключаться или в которую будут создаваться файлы декларации `.d.ts`.

ts

```
{
  "compilerOptions": {
      "declarationDir": ""
  }
}
```

type: `string` **default**: `""` **values**: `path to dir with .d.ts`

[62.29] types

`--types` - декларации, которые размещены в `/node_modules/@types/` и чьи идентификаторы перечислены в массиве, будут доступны глобально. В случае указания пустого массива, глобальный доступ к декларациям будет запрещен. К декларациям, которые запрещены глобально, можно получить доступ только путем импортирования модуля описываемого декларацией.

ts

```
{
  "compilerOptions": {
      "types": [ "node", "express", "rxjs" ]
  }
}
```

type: `string[ ]` **default**: null **values**: `lib identifier`

Важно — все примеры ниже производились после установки декларации. `npm i -D @types/react`

Пример с отключенной опцией

ts

```
// tsconfig.json

{
  "compilerOptions": {

  }
}
```

ts

```
// index.js

class T1 extends React.Component {} // Ok -> global


import {Component} from 'react';

class T2 extends Component {} // Ok -> import
```

Пример с пустым массивом

ts

```
// tsconfig.json

{
  "compilerOptions": {
      "types": [

      ]
  }
}
```

ts

```
// index.js

class T1 extends React.Component {} // Error -> global


import {Component} from 'react';

class T2 extends Component {} // Ok -> import
```

Пример с установленным значением

ts

```
// tsconfig.json

{
  "compilerOptions": {
      "types": [ "react" ]
  }
}
```

ts

```
// index.js

class T1 extends React.Component {} // Ok -> global


import {Component} from 'react';

class T2 extends Component {} // Ok -> import
```

[62.30] typeRoots

`--typeRoots` - ожидает массив путей до директорий с декларациями.

ts

```
{
  "compilerOptions": {
      "typeRoots": [ "./types" ]
  }
}
```

type: `string[ ]` **default**: `null` **values**: `path to .d.ts`

[62.31] allowUnusedLabels

`--allowUnusedLabels` - в случае, если флаг `--allowUnusedLabels` не активен, при выявлении неиспользуемых `label` возникают ошибки.

ts

```
{
  "compilerOptions": {
      "allowUnusedLabels": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.32] noImplicitReturns

`--noImplicitReturns` - функции, имеющие возвращаемый тип, отличный от типа `void`, фактически могут не возвращать значение явно. Другими словами, что бы удовлетворять условиям данной опции, в теле функции должен присутствовать лишь оператор `return`. Но при активной текущей опции, функции будут обязаны возвращать значение явно.

ts

```
{
  "compilerOptions": {
      "noImplicitReturns": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

Пример

ts

```
// noImplicitReturns === false

function f(value: number): number{
 if (value) {
   return; // Ok
 }

 // Ok
}
```

ts

```
// noImplicitReturns === true

function f(value: number): number{
 if (value) {
   return; // Error
 }
}
```

ts

```
// noImplicitReturns === true

function f(value: number): number{
 if (value) {
   return value; // Ok
 }

 // Error
}
```

ts

```
// noImplicitReturns === true

function f(value: number): number{
 if (value) {
   return value; // Ok
 }

 return; // Error
}
```

ts

```
// noImplicitReturns === true

function f(value: number): number{
 if (value) {
   return value; // Ok
 }

 return 0; // Ok
}
```

[62.33] noFallthroughCasesInSwitch

`--noFallthroughCasesInSwitch` - при активной опции в случае, если блок кода `case` имеет код, при этом не имеет выхода из него (`break` или `return`), возникнет ошибка.

ts

```
{
  "compilerOptions": {
```

```
        "noFallthroughCasesInSwitch": false
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

Пример

ts

```ts
function isAdmin( value: string ){
  switch (value) {
      case 'user': // If noFallthroughCasesInSwitch === false then Ok
          if( nodeenv.mode === 'dev' ){
              console.log( `...` );
          }
      case 'moderator':
          return false;
      case 'admin':
          return true;
  }
}
function isAdmin( value: string ){
  switch (value) {
      case 'user': // If noFallthroughCasesInSwitch === true then Error
          if( nodeenv.mode === 'dev' ){
              console.log( `...` );
          }
      case 'moderator':
          return false;
      case 'admin':
          return true;
  }
}
```

[62.34] outFile

`--outFile` - компилятор, при условии, что в качестве модулей указанно `amd` или `system` , будет сохранять все скомпилированные модули в один файл, указанный в качестве значения опции.

ts

```
{
  "compilerOptions": {
      "outFile": ""
  }
}
```

type: `string` **default**: `""` **values**: `path to bundle`

до компиляции

ts

```ts
// T1.ts

export default class T1 {}

// T2.ts

import T1 from './T1';

export default class T2 extends T1 {}

// index.js

import T2 from './T2';

const v1: T2 = new T2();
```

после компиляции

ts

```ts
define("T1", ["require", "exports"], function (require, exports) {
  "use strict";
  Object.defineProperty(exports, "__esModule", { value: true });
  class T1 {
  }
  exports.default = T1;
});
define("T2", ["require", "exports", "T1"], function (require, exports,
T1_1) {
  "use strict";
  Object.defineProperty(exports, "__esModule", { value: true });
  class T2 extends T1_1.default {
  }
  exports.default = T2;
});
define("index", ["require", "exports", "T2"], function (require,
exports, T2_1) {
  "use strict";
  Object.defineProperty(exports, "__esModule", { value: true });
```

```
  const v1 = new T2_1.default();
});
```

[62.35] allowSyntheticDefaultImports

`--allowSyntheticDefaultImports` - позволяет предотвращать ошибки, которые возникают во время сборки по причине несовместимости *SystemJS* и *CommonJS*. Дело в том, что в *ES6* синтаксисе есть возможность экспорта по умолчанию (`export default`), после компиляции которого *CommonJS* испытывает трудности, так как не знает, что такое `default` . Чаще всего проблема возникает в случая, когда разработка ведется с применением одних модулей, а подключаемые библиотеки используют другие.

ts

```
{
  "compilerOptions": {
      "allowSyntheticDefaultImports": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.36] allowUnreachableCode

`--allowUnreachableCode` - если значение данной опции выставлено в `true` , то при обнаружении неиспользуемого кода будет выводиться сообщение об ошибке.

ts

```
{
  "compilerOptions": {
      "allowUnreachableCode": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

Пример

ts

```ts
function f1(value: number): void {
  throw new Error('');

  console.log(''); // Error - Unreachable code detected
}

function f2() {
  return
  {
      value: '' // Error - Unreachable code detected
  }
}
```

[62.37] allowJs

`--allowJs` - в случае, когда код, одна часть которого написана на *TypeScript*, а другая на *JavaScript*, требуется собрать в общий файл, достаточно активировать текущую опцию.

ts

```ts
{
  "compilerOptions": {
      "allowJs": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

Пример

ts

```ts
// sun.js

export default function sum(a, b){
  return a + b;
}
```

```ts
// index.js

import sum from './sum';

let result: number = sum(1, 1);
```

[62.38] reactNamespace

`--reactNamespace` - позволяет установить фабрику рендера `.jsx` синтаксиса.

ts

```ts
{
  "compilerOptions": {
      "reactNamespace": ""
  }
}
```

type: `string` **default**: `""` **values**: `"*"`

Пример

ts

```ts
// index.js

import {jsxFactory} from "jsxFactory";

const div = <div>Hello JSX!</div>;

// after compile

"use strict";
var jsxFactory_1 = require("jsxFactory");
var div = jsxFactory_1.jsxFactory.createElement("div", null, "Hello
JSX!");
```

[62.39] pretty

`--pretty` - раскрашивает в разные цвета выводимые в консоль сообщения, делая их более понятными.

ts

```
{
  "compilerOptions": {
      "pretty": true
  }
}
```

type: `boolean` **default**: `true` **values**: `true` , `false`

[62.40] moduleResolution

`--moduleResolution` - позволяет конкретизировать поведение модулей.

ts

```
{
  "compilerOptions": {
      "moduleResolution": "node"
  }
}
```

type: `string` **default**: module === `AMD` | `System` | `ES6` ? `classic` : `node` **values**: `classic` , `node`

[62.41] exclude

`exclude` - в обычной ситуации под компиляцию рекурсивно попадают все файлы, которые включает в себя директория, содержащая `tsconfig.json`. Решением этой проблемы является свойство `exclude`, определенное в корне конфигурационного файла. В качестве его значения выступает массив с путями к файлам или директориям, которые следует исключить из компиляции.

ts

```
{
  "exclude": [

  ]
}
```

type: `string[ ]` **default**: `null` **values**: `path to file or dir`

[62.42] noEmitHelpers

`--noEmitHelpers` - после компиляции файлов с расширением `.ts` каждый скомпилированный файл `.js` (модуль) содержит, если в этом есть необходимость, вспомогательный код (проще говоря, helpers), который помогает решить проблему совместимости версий *ES*. В большей степени этот код, находящийся в разных файлах (модулях), идентичен. Простыми словами, helpers повторяются от файла к файлу. Компилятор вынужден добавлять повторяющийся код, что бы гарантировать работу каждого модуля по отдельности. Но если модули собираются в одну общую сборку, активация текущего флага укажет компилятору, что helpers нужно вынести в отдельный модуль, который будет доступен в зависящих от него частях кода. Это поможет избавиться от дублирования кода и сократит размер собранного файла.

ts

```
{
  "compilerOptions": {
```

```
        "noEmitHelpers": false
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.43] newLine

`--newLine` - по умолчанию, новая строка обозначается `\r\n` на *Windows* и `\r` на *nix* системах. Для переопределения поведения используется данная опция.

ts

```
{
    "compilerOptions": {
        "newLine": "LF"
    }
}
```

type: `string` **default**: `platform specific` **values**: `CRLF` , `LF`

[62.44] inlineSourceMap

`--inlineSourceMap` - активная опция приводит к тому, что *source maps* записываются *inline* в `.js` файлы, а не выносятся в отдельные файлы `.map.js` .

ts

```
{
    "compilerOptions": {
        "inlineSourceMap": false
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.45] inlineSources

`--inlineSources` - при активной опции, источник *source map* включается в файл вместе с *source map*. Работает в паре с опцией `--inlineSourceMap` .

ts

```
{
  "compilerOptions": {
      "inlineSources": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.46] noEmitOnError

`--noEmitOnError` - несмотря на ошибки в *TypeScript* коде, компилятор все равно генерирует `.js` файлы. Для того, что бы компилятор генерировал файлы `.js` только в случае успешной компиляции, нужно активировать данную опцию.

ts

```
{
  "compilerOptions": {
      "noEmitOnError": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.47] noEmit

`--noEmit` - при активной опции информация о компиляции перестает выводиться.

ts

```
{
  "compilerOptions": {
      "noEmit": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.48] charset

`--charset` - устанавливает формат входных файлов.

ts

```
{
  "compilerOptions": {
      "charset": ""
  }
}
```

type: `string` **default**: `utf8` **values**: `*`

[62.49] diagnostics

`--diagnostics` - выводит диагностическую информацию.

ts

```
{
  "compilerOptions": {
      "diagnostics": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.50] declaration

`--declaration` - генерирует файлы декларации `.d.ts` из `.ts` файлов.

ts

```
{
  "compilerOptions": {
      "declaration": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.51] downlevelIteration

`--downlevelIteration` - при активной опции становится возможно использовать итераторы при компиляции в версии ниже `ES6`. Помимо самих итераторов, становятся доступны и нововведения (`for...of`, `Array Destructuring`, `Spread` и т.д.), которые построены с их использованием.

ts

```
{
  "compilerOptions": {
      "downlevelIteration": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.52] emitBOM

`--emitBOM` - Извлекает *маркер последовательности байтов UTF-8* (BOM) в начале выходных файлов.

ts

```
{
  "compilerOptions": {
      "emitBOM": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.53] emitDecoratorMetadata

`--emitDecoratorMetadata` - ...

ts

```
{
  "compilerOptions": {
      "emitDecoratorMetadata": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.54] forceConsistentCasingInFileNames

`--forceConsistentCasingInFileNames` - запрещает несогласованные ссылки на один и тот же файл.

ts

```
{
  "compilerOptions": {
      "forceConsistentCasingInFileNames": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.55] help (h)

`--help` или `-h` - выводит список доступных опций.

ts

```
tsc --help
tsc -h
```

[62.56] importHelpers

`--importHelpers` - импортирует таких помощников (helpers), как `__extends`, `__rest` и т.д.

ts

```
{
  "compilerOptions": {
      "importHelpers": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.57] isolatedModules

`--isolatedModules` - транслирует каждый файл, как отдельный модуль.

ts

```
{
  "compilerOptions": {
      "isolatedModules": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.58] listEmittedFiles

`--listEmittedFiles` - выводит список сгенерированных файлов.

ts

```
{
  "compilerOptions": {
      "listEmittedFiles": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.59] listFiles

`--listFiles` - выводит список участвующих в компиляции файлов.

ts

```
{
  "compilerOptions": {
      "listFiles": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.60] sourceRoot

`--sourceRoot` - в качестве значения принимает базовый путь до директории, в которой лежат исходники `.ts` , необходимые для ассоциации с *source map*.

ts

```
{
  "compilerOptions": {
      "sourceRoot": ""
  }
}
```

type: `string` **default**: `""` **values**: `path to source  .ts dir`

[62.61] mapRoot

`--mapRoot` - место, откуда будут браться файлы `.map.js` .

ts

```
{
  "compilerOptions": {
      "mapRoot": ""
  }
}
```

type: `string` **default**: `""` **values**: ``

[62.62] maxNodeModuleJsDepth

`--maxNodeModuleJsDepth` - максимальная глубина поиска зависимостей в `/node_modules` и загрузки файлов `.js` . Работает только с активной опцией `--allowJs` .

ts

```
{
  "compilerOptions": {
      "maxNodeModuleJsDepth": 0
  }
}
```

type: `number` **default**: `0` **values**: `0...n`

[62.63] project (p)

`--project` или `-p` - с помощью этого флага можно указать как путь до директории, которая содержит `tsconfig.json` , так и на конкретный `tsconfig.json` файл.

ts

```
tsc --project
tsc -p ./configs/tsconfig.es6.json
```

type: `string` **default**: `` **values**: `path to tsconfig.json`

[62.64] init

`--init` - создает новый `tsconfig.json` со всеми доступными опциями, большинство из которых закомментировано, дабы создать оптимальную конфигурацию.

sh

```
tsc --init
```

[62.65] version (v)

`--version` или `-v` - выводит информацию о текущей версии *TypeScript*.

sh

```
tsc --version
tsc -v
```

[62.66] watch (w)

`--watch` или `-w` - запускает компилятор в режиме наблюдения за изменением файлов. При каждом изменении отслеживаемых файлов компиляция будет запущена автоматически.

sh

```
tsc --watch
tsc -w
```

[62.67] preserveSymlinks

`--preserveSymlinks` - текущая опция демонстрирует поведение, идентичное реализуемому в *NodeJS* с активным флагом `--preserve-symlinks`. При активной опции символические ссылки на модели (modules) и пакеты (packages) разрешаются относительно файла символической ссылки, а не относительно пути, к которому разрешается символическая ссылка. Кроме того, поведение при активной текущей опции противоположно поведению, предоставляемому *Webpack* с помощью флага со схожим по смыслу названием `resolve.symlinks`.

ts

```
{
  "compilerOptions": {
      "preserveSymlinks": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.68] strictFunctionTypes

`--strictFunctionTypes` - при активной опции параметры функций начинают сравниваться по контрвариантным правилам, в, то время, как при не активной опции, сравнения производятся по бивариантным правилам.

ts

```
{
  "compilerOptions": {
      "strictFunctionTypes": false
```

```
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.69] locale

`--locale` - позволяет указать один из заданных языков для вывода диагностических сообщений.

ts

```
{
  "compilerOptions": {
      "locale": "en" | "cs" | "de" | "es" | "fr" | "it" | "ja" | "ko" |
"pl" | "pt-BR" | "ru" | "tr" | "zh-CN" | "zh-TW" |
  }
}
```

type: `string` **default**: `platform specific` **values**: English (US): `en` ,Czech: `cs` ,German: `de` ,Spanish: `es` ,French: `fr` ,Italian: `it` ,Japanese: `ja` ,Korean: `ko` ,Polish: `pl` ,Portuguese(Brazil): `pt-BR` ,Russian: `ru` ,Turkish: `tr` ,Simplified Chinese: `zh-CN` ,Traditional Chinese: `zh-TW`

[62.70] strictPropertyInitialization

`--strictPropertyInitialization` - при активной опции в случае, когда в классе присутствуют поля, не инициализированные в момент создания или в конструкторе, возникает ошибка. Более подробно данная тема раскрыта в главе “Классы - Definite Assignment Assertion Modifier”.

ts

```
{
  "compilerOptions": {
```

```
        "strictPropertyInitialization": false
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

ts

```
class Identifier {
    public a: number = 0; // Ok, инициализация при объявлении
    public b: number; // Ok, инициализация в конструкторе
    public c: number | undefined; // Ok, явное указание принадлежности к
типу Undefined
    public d: number; // Error, инициализация отсутствует
    constructor() {
        this.b = 0;
    }
}
```

[62.71] esModuleInterop

`--esModuleInterop` - с активной опцией сгенерированный код таких модулей формата *CommonJS/AMD/UMD* больше походит на код, сгенерированный *Babel*.

ts

```
{
    "compilerOptions": {
        "esModuleInterop": false
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.72] emitDeclarationsOnly

`--emitDeclarationsOnly` - данная опция указывает компилятору, что нужно генерировать только файлы декларации с расширением `.d.ts` и пропускать файлы с расширением `.js` и `.jsx`. Такое поведение может быть полезно, если код, помимо компилятора *TypeScript*, компилируется ещё и с помощью *Babel*.

ts

```
{
  "compilerOptions": {
      "emitDeclarationsOnly": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.73] resolveJsonModule

`--resolveJsonModule` - данная опция, при активной опции `--esModuleInterop` и опции `--module`, установленной в `commonjs`, позволяет в среде *NodeJS* полноценно взаимодействовать с *json*.

ts

```
{
  "compilerOptions": {
      "resolveJsonModule": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

ts

```
// file config.json

{
  "name": "",
  "age": 0
}

// file index.js


import config from "./config.json";

config.name = 'name'; // Ok
config.name = 0; // Error


// tsconfig.json


{
   "compilerOptions": {
       "module": "commonjs",
       "resolveJsonModule": true,
       "esModuleInterop": true
   }
}
```

[62.74] declarationMap

`--declarationMap` - при совместном использовании с активной опцией `--declaration` заставляет компилятор, помимо `.d.ts`, также генерировать `.d.ts.map`, которые позволяют при переходе к определению (go to definition) направлять в файл `.ts`, а не `.d.ts`.

ts

```
{
  "compilerOptions": {
      "declarationMap": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.75] strictBindCallApply

`--strictBindCallApply` - текущий флаг, входящий в группировку `--strict`, активирует проверку вызова таких методов, как `apply`, `call` и `bind`. Это стало возможным благодаря добавлению двух новых типов, `CallableFunction` и `NewableFunction`, которые содержат обобщенное описание методов `apply`, `call` и `bind`, как для обычных функций, так и для функций конструкторов соответственно.

ts

```
{
  "compilerOptions": {
      "strictBindCallApply": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

ts

```
function f(p0: number, p1: string){}


f.call(null, 0, ''); // ok
f.call(null, 0, 0); // error
f.call(null, 0); // error
f.call(null, 0, '', 0); // error

f.apply(null, [0, '']); // ok
f.apply(null, [0, 0]); // error
f.apply(null, [0]); // error
f.apply(null, [0, '', 0]); // error

f.bind(null, 0, ''); // ok
f.bind(null, 0, 0); // error
f.bind(null, 0); // ok
f.bind(null, 0, '', 0); // ok
```

[62.76] showConfig

`--showConfig` - при активном текущем флаге компилятор `tsc` во время компиляции выведет в консоль содержимое конфигурационного файла `tsconfig.json`, разрешенного с учетом механизма расширения (`extends`), если таковой механизм имеет место быть. Этот функционал может быть очень полезным при отладке.

ts

```
{
  "compilerOptions": {
      "showConfig": false
  }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.77] build

`--build` - данный флаг указывает компилятору `tsc` , что проект нужно собирать как проект, использующий ссылки на другие проекты. Подробнее об этом можно узнать из главы Сборка с использованием ссылок на проекты. Также флаг `--build` может использоваться в сочетании со специфичными только для него флагами `--verbose` , `--dry` , `--clean` , `--force` , а также с флагом `--watch` .

type: `string` **default**: `` **values**: `paths to tsconfig.json or dir with tsconfig.json`

[62.78] verbose

`--verbose` - текущий флаг указывает компилятору выводить более подробный отчет при инкрементальной сборке проекта. Используется только совместно с флагом `--build` .

[62.79] dry

`--dry` - при указании текущего флага сборка будет выполнена без порождения выходных файлов. Данный флаг полезно использовать совместно с флагом `--verbose` . Используется только совместно с флагом `--build` .

[62.80] clean

`--clean` - удаляет выходные файлы, соответствующие заданным входным. Используется только совместно с флагом `--build` .

[62.81] force

`--force` - принудительно выполняет не инкрементальную сборку. Используется только совместно с флагом `--build`.

[62.82] incremental

`--incremental` - флаг, при активации которого, после первой компиляции проекта, в заданной атрибутом `outDir` директории создается файл `.tsbuildinfo`, который хранит метаинформацию об изменении файлов, что позволяет производить ускоренные инкрементальные сборки при всех последующих запусках компилятора.

ts

```
{
    "compilerOptions": {
        "incremental": true,
        "outDir": "./dest"
    }
}
```

В случае, когда имя выходного файла задается с помощью флага `--outFile`, имя генерируемого файла `.tsbuildinfo` будет включать в себя название выходного файла (`.client.tsbuildinfo` для файла `client.js` и `.server.tsbuildinfo` для `server.js` соответственно).

Примечание: создатели *TypeScript* заранее предупреждают, что генерируемые файлы `.tsbuildinfo` не предназначены для использования сторонними библиотеками, так как их определение не будет обладать совместимостью от версии к версии.

Кроме того, с помощью флага `--tsBuildInfoFile` можно задать место сохранения файла `.tsbuildinfo`.

ts

```
{
    "compilerOptions": {
        "incremental": true,
        "tsBuildInfoFile": "./buildinfo",
        "outDir": "./dest"
    }
}
```

[62.83] tsBuildInfoFile

`--tsBuildInfoFile` - флаг, с помощью которого указывается место сохранения файла `.tsbuildinfo`, генерирующегося при активной опции `--incremental` и служащего для хранения метаинформации, призванной ускорить последующие сборки.

ts

```
{
    "compilerOptions": {
        "incremental": true,
        "tsBuildInfoFile": "./buildinfo",
    }
}
```

[62.84] allowUmdGlobalAccess

`--allowUmdGlobalAccess` - при активном текущем флаге становится возможным обращение к глобальным определениям из модулей.

ts

```
{
    "compilerOptions": {
        "allowUmdGlobalAccess": false,
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

ts

```ts
// allowUmdGlobalAccess === false

import * as Rx from 'rxjs';

const ref = React.createRef(); // Error, обращение к глобальным
переменным в модулях недопустимо
```

ts

```ts
// allowUmdGlobalAccess === true

import * as Rx from 'rxjs';

const ref = React.createRef(); // Ok, доступ к глобальному определению
из модуля
```

[62.85] disableSourceOfProjectReferenceRedirect

`--disableSourceOfProjectReferenceRedirect` - при использовании механизма ссылок на проекты активация данного флага говорит компилятору, что в качестве информации о типах следует использовать файлы декларации `.d.ts` , а не исходные файлы проекта. Активация данного флага способна повысить производительность сборки, но вносит некоторую специфику, поэтому уместна лишь на больших проектах. Более подробно читайте об этом в главе, посвященной использованию ссылок на проекты.

ts

```ts
{
    "compilerOptions": {
        "disableSourceOfProjectReferenceRedirect": false,
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.86] useDefineForClassFields

`--useDefineForClassFields` - данный флаг активирует новое поведение генерации конечного кода, доступное с версии `v3.7` и предназначенное для предотвращения переопределения свойств при механизме наследования.

[Важно] Начиная с версии `4.0` логика компилятора *TypeScript* подразумевает не переопределяемое поведение равнозначное поведению при активном текущем флаге.

ts

```
{
    "compilerOptions": {
        "useDefineForClassFields": false,
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.87] importsNotUsedAsValues

`--importsNotUsedAsValues` - задает стратегию используемую компилятором для разрешения зависимостей модуля путем уточнения формы импорта и экспорта. Более подробно о текущем флаге можно прочесть в главе "Импорт и экспорт только типа и флаг –importsNotUsedAsValues".

ts

```
{
    "compilerOptions": {
        "importsNotUsedAsValues": "remove",
    }
}
```

type: `string` **default**: `remove` **values**: `remove` , `preserve` , `error`

[62.88] assumeChangesOnlyAffectDirectDependencies

`--assumeChangesOnlyAffectDirectDependencies` - в режиме `--watch` + `--incremental` активация данной опции позволяет компилятору отказаться от перепроверок\перестраивания файлов, которые на основе метаинформации расцениваются затронутыми. Вместо этого будут перепроверяться\перестраиваться только непосредственно изменённые файлы и файлы их импортирующие.

Представьте, что `fileA.ts` импортирует `fileB.ts`, который импортирует `fileC.ts`, который импортирует `fileD.td`.

При активном режиме `--watch` изменения в файле `fileD.ts` означает, что как минимум будут проверены `fileC.ts`, `fileB.ts` и `fileA.ts`. При активной опции `--assumeChangesOnlyAffectDirectDependencies` проверке подвергнется лишь `fileA.ts` и `fileB.ts`.

json

```
// tsconfig.json

{
    "watchOptions": {
        "assumeChangesOnlyAffectDirectDependencies": "false"
    }
}
```

type: `boolean` **default**: `false` **values**: `true`, `false`

[62.89] watchFile

`--watchFile` - стратегия наблюдения за отдельными файлами.

json

```
// tsconfig.json

{
    "watchOptions": {
        "watchFile": "useFsEvents"
    }
}
```

type: `string` **default**: `useFsEvents` **values**: `fixedPollingInterval`, `priorityPollingInterval`, `dynamicPriorityPolling`, `useFsEvents`, `useFsEventsOnParentDirectory`

описание

- `fixedPollingInterval` : Проверять каждый файл на наличие изменений несколько раз в секунду с фиксированным интервалом.
- `priorityPollingInterval` : Проверять каждый файл на наличие изменений несколько раз в секунду, но использовать эвристику для проверки файлов определенных типов реже, чем других.
- `dynamicPriorityPolling` : Использовать динамическую очередь, в которой менее часто изменяемые файлы будут проверяться реже.
- `useFsEvents` [ПО УМОЛЧАНИЮ]: Пытаться использовать собственные события операционной системы / файловой системы для изменения файлов.
- `useFsEventsOnParentDirectory` : Пытаться использовать собственные события операционной системы/файловой системы для прослушивания изменений в каталогах, содержащих файл. Это может использовать меньше файловых наблюдателей, но также быть менее точным.

[62.90] watchDirectory

`--watchDirectory` - стратегия наблюдения за целыми деревьями каталогов в системах, в которых отсутствует рекурсивная функция наблюдения за файлами.

json

```
// tsconfig.json

{
```

```
    "watchOptions": {
        "watchDirectory": "useFsEvents"
    }
}
```

type: `string` **default**: `useFsEvents` **values**: `fixedPollingInterval` , `dynamicPriorityPolling` , `useFsEvents`

описание

- `fixedPollingInterval` : Проверять каждый каталог на наличие изменений несколько раз в секунду с фиксированным интервалом.
- `dynamicPriorityPolling` : Использовать динамическую очередь, в которой менее часто изменяемые каталоги будут проверяться реже.
- `useFsEvents` [ПО УМОЛЧАНИЮ]: Пытаться использовать собственные события операционной системы / файловой системы для изменений каталога.

[62.91] fallbackPolling

`--fallbackPolling` - при использовании событий файловой системы этот параметр определяет стратегию опроса, которая используется, когда в системе заканчиваются собственные наблюдатели файлов и/или не поддерживаются собственные средства просмотра файлов.

json

```
// tsconfig.json

{
    "watchOptions": {
        "fallbackPolling": "dynamicPriorityPolling"
    }
}
```

type: `string` **default**: `useFsEvents` **values**: `fixedPollingInterval` , `dynamicPriorityPolling` , `priorityPollingInterval`

описание

- `fixedPollingInterval` : Проверять каждый файл на наличие изменений несколько раз в секунду с фиксированным интервалом.
- `dynamicPriorityPolling` : Использовать динамическую очередь, в которой менее часто изменяемые файлы будут проверяться реже.
- `priorityPollingInterval` : Проверять каждый файл на наличие изменений несколько раз в секунду, но использовать эвристику для проверки файлов определенных типов реже, чем других.

[62.92] synchronousWatchDirectory

`--synchronousWatchDirectory` - отключить отложенное наблюдение за каталогами.

json

```
// tsconfig.json

{
    "watchOptions": {
        "synchronousWatchDirectory": "false"
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.93] noUncheckedIndexedAccess

`--noUncheckedIndexedAccess` - при активной текущей опции обращаться к динамическим членам объекта разрешается только после подтверждения их существования, а также совместно с такими механизмами как опциональный оператор `!.` или оператор опциональной последовательности `?.` .

json

```json
// @filename: tsconfig.json

{
    "watchOptions": {
        "noUncheckedIndexedAccess": "true"
    }
}
```

type: `boolean` **default**: `true` **values**: `true`, `false`

json

```json
// @filename: tsconfig.json

{
    "compilerOptions": {
        "noUncheckedIndexedAccess": false
    }
}
```

ts

```ts
type T = {
    [key: string]: number | string;
}

function f(p: T) {
    /**
     * Обращение к несуществующим полям
     */
    p.bad.toString(); // Ok -> Ошибка времени исполнения
    p[Math.random()].toString(); // Ok -> Ошибка времени исполнения
}
```

json

```json
// @filename: tsconfig.json

{
    "compilerOptions": {
        "noUncheckedIndexedAccess": true
    }
}
```

ts

```ts
type T = {
  [key: string]: number | string;
}
```

```
function f0(p: T) {
  /**
   * Обращение к несуществующим полям
   */
  p.bad.toString(); // Error -> Object is possibly 'undefined'.ts(2532)
  p[Math.random()].toString(); // Error -> Object is possibly
'undefined'.ts(2532)


  // Проверка наличия поля bad
  if("bad" in p){
      p.bad?.toString(); // Ok
  }

  // Использование опционального оператора
  p[Math.random()]!.toString(); // Ok -> ошибка во время выполнения

  p[Math.random()]?.toString();  // Ok -> Ошибка не возникнет
}

function f1(array: string[]) {
    for(let i = 0; i < array.length; i++){
        array[i].toString(); // Error -> Object is possibly 'undefined'.
    }
}
```

[62.94] noPropertyAccessFromIndexSignature

`--noPropertyAccessFromIndexSignature` - активирует поведение запрещающее обращение через точечную нотацию к динамическим членам объекта определяющего строковую индексную сигнатуру,

json

```
// @filename: tsconfig.json

{
    "compilerOptions": {
        "noPropertyAccessFromIndexSignature": "false"
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

ts

```ts
type Settings = {
    env?: string[]; // определение необязательного предопределенного
поля

    [key: string]: any; // определение динамических полей
}


function configurate(settings: Settings){
    //---------------------------
    // динамическое поле
    if(settings.envs){ // Ошибка при активном флаге и Ok при не активном

    }
    if(settings['envs']){ // Ok при любом значении флага

    }

    //---------------------------
    // предопределенное поле
    if(settings.env){ // Ok [1]

    }
    if(settings['env']){ // Ok при любом значении флага

    }
}
```

[62.95] explainFiles

`--explainFiles` - команда, позволяющая выводить информацию о зависимостях проекта не только в консоль, но и файл, или даже открывать в *visual studio code*.

bash

```bash
// вывод в файл
tsc --explainFiles > explanation.txt

// вывод в редактор vsc
tsc --explainFiles | code -
```

[62.96] noImplicitOverride

`--noImplicitOverride` - активирует механизм предотвращающий объявление суперклассом членов уже объявленных в его потомках.

json

```
// @filename: tsconfig.json

{
    "compilerOptions": {
        "noImplicitOverride": "false"
    }
}
```

type: `boolean` **default**: `false` **values**: `true` , `false`

[62.97] useUnknownInCatchVariables

`--useUnknownInCatchVariables` - при активном флаге единственный параметр блока `catch` принадлежит к типу `unknown` . Иначе, к типу `any` . Данный флаг входит в состав группировки `strict` .

json

```
// @filename: tsconfig.json

{
    "compilerOptions": {
        "useUnknownInCatchVariables": "true"
    }
}
```

ts

```
try {

}catch(error){
    /**
     * useUnknownInCatchVariables === true - error is unknown
     * useUnknownInCatchVariables === false - error is any
     */
}
```

type: `boolean` **default**: `true` **values**: `true` , `false`

[62.98] exactOptionalPropertyTypes

`--exactOptionalPropertyTypes` - при активном флаге запрещается присваивать значение `undefined` необязательным полям объекта без явной принадлежности к типу `undefined` .

json

```
// @filename: tsconfig.json

{
    "compilerOptions": {
        "exactOptionalPropertyTypes": "false"
    }
}
```

ts

```
// exactOptionalPropertyTypes = true

type T = {
    a: number;
    b?: string;
}

let o: T = {
    a: 5,
    b: undefined // Error -> Type 'undefined' is not assignable to type
'string'.ts(2322)
};
```

type: `boolean` **default**: `false` **values**: `true`, `false`

Конец!